Design and Implementation of an Attribute Manager for Conditional and Distributed Graph Transformation

Boris Melamed

Master's Thesis
at the Computer Science Department of the
Technical University of Berlin

Abstract

This thesis describes an implemented attribute component that can be used for distributed and attributed graph transformation with application conditions. Realized functionality includes managing of attribute tuples, where matching and transforming of attributes are performed. The provided exemplary attribute handler interprets Java expressions. Rule variables and parameters as well as attribute application conditions are considered. Attribute interfaces can be generated, realizing distribution concepts. New attribute type handlers can be added to the system, possibly including custom editors. Different views on attributes (hiding, changing the order) are supported. Supplied editors allow for entering and manipulating of attribute types and values that are immediately checked for correctness. They can be replaced or extended by other GUI elements. Although developed as part of the algebraic graph transformation system AGG, the attribute component can be employed within other specification tools, such as in place-transition-systems (Petri nets).

Zusammenfassung

Diese Arbeit beschreibt eine implementierte Attributkomponente, die für verteilte, attributierte Graphtransformation mit Anwendungsbedingungen verwendet werden kann. Realisierte Funktionalität umfaßt die Verwaltung von Attributtupeln, wobei Ansatz und Transformation möglich sind. Der exemplarische Attribut-Handler verarbeitet Java-Ausdrücke. Regelvariablen und –parameter sowie Anwendungsbedingungen werden berücksichtigt. Attribut-Interfaces können erzeugt werden, womit Verteilung möglich wird. Das System kann um neue Attribut-Handler erweitert werden, einschließlich spezieller Editoren. Verschiedene Sichten auf Attribute (Ausblenden, Ändern der Reihenfolge) werden unterstützt. Angebotene Editoren erlauben das Eingeben und Ändern von Attributtypen und –werten, die dann sofort auf Korrektheit überprüft werden. Sie können durch andere GUI- Elemente ersetzt oder erweitert werden. Obwohl als Teil des algebraischen Graphtransformationssystems AGG entwickelt, kann die Attributkomponente innerhalb anderer Spezifikationstools verwendet werden, so z.B. in Stellen- Transitionsnetzen (Petrinetze).

Contents

Chapter 1

Introduction

Upon giving some motivation for attributed, distributed graph transformation, the goals of this thesis are briefly described and its structure is given.

1.1 Why Graph Transformation?

Whenever we describe complex matters, we are tempted to use graphical notation. In computer science, when creating models of reality and designing systems to perform accordingly, the degree of complexity can be quite high. This explains the popularity of diagrams in software and database specifications. Indeed, every software development method comes with its own set of diagrams. The expressiveness of class diagrams contributed much to the overwhelming success of object orientation.

However, in most representational schema, there's a clear gap between the static and the dynamic aspects of specified systems. Depending on perspective, various kinds of diagrams are used, and the connection between entities in different notations is often lost.

Graph transformation preserves the connection between graphical descriptions of static and dynamic aspects of a system. Rules, describing the functional/dynamic behavior of a graph transformation system, make use of the static graph notation itself. *Algebraic* graph transformation offers a solid mathematical foundation with formal semantics [Ehr79, Löw93]. It allows e.g. for consistency and termination proofs by applying algebraic and category methods. This suggests the impressive prospect of combining the intuitive expressiveness of graphs with automatic processing and support by a machine.

Therefore, graph transformation appears to be quite attractive as a formal specification technique. However, for it go be of practical use, concepts of attributes were added [LKW93], [HMTW95], [CL95]. They allow for textual and other non-graphical information to be integrated into graphs. Further extensions of the formalism include rule application conditions ([Koc97]) and distribution concepts ([Tae96]).

1.2 Goals

Then, the interested reader might ask, what prevents the average software developer from embracing graph transformation and forgetting everything else? For a specification method to be accepted, it needs supporting tools. Assembler programmers weren't tempted to switch to Fortran, Cobol and Pascal until they were offered the respective compilers. Object orientation becomes attractive when effective browsers for monitoring classes and their methods across the inheritance trees are available. Likewise, specification by graph transformation needs corresponding tools: a graphical editor where graphs and rules can be entered and checked for consistency, a transformation machine that matches and applies the rules, stepwise debugging etc.

Existing tools have not yet found users outside the academic community. The most impressive among them so far is probably PROGRES [Zün95, Sch97]. It offers the entering of graph rules, their automatic application and even the generating of stand-alone prototypes. However, it is not capable of modeling distributed systems, and it isn't based on algebraic formal semantics.

The work on an *Algebraic Graph Grammar* system AGG started as early as 1991 [Bey91, LB93]. The result was a graph editor which offered limited transformation support. An attempt to extend that system [Win95] made clear that re-designing was necessary, with a stress on portability and modularity.

Therefore, several system components are being developed simultaneously, communicating through slim, clearly defined interfaces:

Graph Basis Graph model: internal graph and rule structures and their organization [Rud96, Rud97];

Attribute Component Input, administration and transformation of attributes [Mel97], including application conditions, rule variables and parameters and distribution;

Transformation The Colimit library [Wol97] is used for graph transformation. Its correctness implicitly proofs the system's adherence to the theory of algebraic graph transformation;

Application Match Expressed as a Constraint Satisfaction Problem [Pro93, Dec94], the match search keeps the actual algorithm from being bound to a concrete graph model [Rud97].

Graphical Environment A graphical user interface (GUI), offering an interactive input of graphs and rules as well as a transformation control environment.

The evolving tool offers the possibility of actually applying algebraic graph transformation in formal specifications. The system shall then be extended to include modeling of distribution and negative application conditions.

The goal of this thesis is the design and implementation of the attribute component within this framework. Its structure consists of one attribute manager and an arbitrary number of attribute handlers. The attribute manager provides the connection between the graph component and typed tuples (vectors) of attributes. These tuples are managed by the attribute manager, and each tuple element represents an attribute value whose type and functionality is determined by its attribute handler. All necessary interfaces are specified, and the attribute manager as well as an exemplary attribute handler are implemented. This handler interprets expressions following the syntax of the Java computer language, enabling integrated execution of arbitrary Java programs within graph transformation.

If one then wishes to add further attribute types to the attribute component, it can be done so in one of the following two ways:

1. Registering further handlers with the attribute manager;
2. Writing appropriate Java classes for desired functionality and referring to them from within Java expressions.

The attribute manager provides rule variables, rule parameters which can be entered interactively when a rule is applied, application conditions, inheritance of attribute types and the possibility of binding attribute types and values to interfaces, thereby enabling distributed objects to be modeled.

Three main cases of using the attribute component are considered:

1. The graph transformation system is not "interested" in the attribute values. It lets the attribute component manage the input, and match and transform the attributes. It just tells the attribute manager when to do so and with which objects. It is satisfied with the supplied default attribute editors, merely integrating them into its own graphical user interface (GUI) environment.
2. The implemented GUI can be easily replaced by another one altogether, using the attribute manager's corresponding interface. Attribute handlers can supply their own value and customization editors.
3. For cases when the attribute values are used by the graph transformation system itself, e.g. when they define the shape of nodes or contain hints used by the match search heuristics, an interface allows non-interactive introspection and manipulation of values.

To simplify the usage of the attribute component, three corresponding interface abstractions (facades) are also provided. The applicability of a system is improved when its interface contains just the necessary methods.

1.3 Structure

Chapter 2 introduces the theoretical concepts. After specifying the requirements (chapter 3) and describing the employed tools and methods (chapter 4), the interface design is explained in chapter 5. Chapter 6 relates how to integrate the attribute component and how its services can be used in other systems, depending on the usage aspects as depicted above. This chapter can be also seen as a manual for developers of systems that use the attribute component. Chapter 7 gives an overview as well as some details of the implementation and serves developers who want to maintain the implemented system. Chapter 8 consists of a user manual for supplied attribute editors. Chapter 9 sums up the results and discusses some perspectives. The appendix contains a detailed documentation of all interfaces and their implementing classes in Java.

Chapter 2

Distributed and Attributed Graph Transformation

In graph transformation systems, rules are applied to the working graph in order to transform it according to a given grammar. A rule can be applied when its left side can match the working graph, meaning an inclusion from the left rule side into the working graph can be found such that the application conditions of the rule are fulfilled.

Eventually, each graph object of the source graph (left rule side) has to have a matching graph object in the target graph (working graph), such that also the object's attributes match. Whenever the match-searching algorithm decides to match two graphical objects

Each section of this chapter presents the theoretical concepts of an aspect of algebraic distributed and attributed graph transformation and then goes on to describe how these are mapped into an appropriate tool. The last section gives just an overview of Java expressions accepted by the standard attribute handler.

2.1 Attributed Graph Transformation

After a short overview of existing approaches to attributed graph transformation, reasons are given for the choice of the eventually utilized formal approach. The formal definition is presented and characteristic examples are discussed. Then the chosen tool realization approach is presented.

2.1.1 Existing Approaches

Nearly all graph transformation approaches can have node and/or edge labels that can or cannot be changed during transformation. The non-changeable labels represent node and edge types and help to structure the transformation process. The changeable labels are used for storing data with a graphical object (node or edge). These labels are usually called attributes.

There are different concepts of graph transformation. In category grammars [Sch93], nodes and edges can be labeled by objects of arbitrary categories. The resulting graph grammar approach can transform graphs as well as labels, but the rules must be injective within this approach.

In [Sch92], a general approach for graph labeling is presented, formalized by comma categories. It also allows labeling of nodes and edges by elements of arbitrary categories. This general approach includes labeling by sets, partial orders or total algebras, as shown in [Sch92].

All the other attribute concepts follow an algebraic approach. This means that all attributes in a graph belong to an algebra that describes some kind of data types. This algebra remains unchanged throughout the transformation process. There are approaches which use total algebras for attributes, as in [Sch92] for the Double-Pushout-Approach and [LKW93] for the Single-Pushout-Approach. Further works extend these concepts towards partial attribute algebras [CL95] and even more general, give up the strict separation of graph- and data structure, in order to obtain a partial algebra transformation [Wag97].

Among the existing concepts of graph transformation, the *algebraic graph transformation* offers a solid mathematical foundation ([Ehr79], [Löw93]). Applying algebraic and category methods, general characteristics within graph transformation can be proven, for example in the following areas:

- Parallel and sequential composition of rules;
- Termination in non-deterministic graph transformation;
- Generation of dynamic application conditions from static consistency conditions;
- Consistency in *distributed* graph transformation.

The approach utilized by this thesis follows the concept in [CL95] since it allows the whole expressiveness of graph transformation (in the Single-Pushout-Approach), as well as a flexible mechanism for describing data and data operations by partial algebras. Both parts are strictly separated, each describing different system aspects. This is reflected by the component-wise construction of pushouts for graphs and for data algebras.

2.1.2 Formal Definitions

The following briefly presents the attribute concepts as described in [CL95].

Definition 2.1 (Attributed graph signature) *An attributed graph signature $AGS = GS + DS + ATTROP$ consists of a graph signature $GS = (S_{GS}, OP_{GS})$, which has only* *unary operations, a signature $DS = (S_{DS}, OP_{DS})$, called data signature, and an $S_{GS} \times S_{DS}$ - indexed family ATTROP of operation sets, the attributing operations.*

Definition 2.2 (AGS-Algebra) *An algebra A wrt. an attributed graph signature $AGS = GS + DS + ATTROP$ is given by a total GS-Algebra A_{GS}, a partial DS-Algebra A_{DS} and a family of functions $op^A : A_s^{GS} \to A_{s'}^{DS}$ for all $op : s \to s' \in ATTROP$.*

Given another AGS-Algebra B, then an AGS-morphism h: $A \to B$ is defined by a partial homomorphism $h^{GS} : A_{GS} \to B_{GS}$ and a total homomorphism $h^{DS} : A_{DS} \to B_{DS}$ such that $h^{DS}(op^A(x)) = op^B(h^{GS}(x))$ for all $op : s \to s' \in ATTROP$ and all $x \in dom(h^{GS})_s$.

Since the total GS-algebras with partial GS-morphisms form a category $\underline{\underline{GS^P}}$ [Löw93], and the partial DS-algebras with total DS-morphisms also form a category [Bur86], it's easy to show that the AGS-algebras with AGS-morphisms form a category $\underline{\underline{AGS}}$.

Definition 2.3 (Rule and match) *A rule $p : L \xrightarrow{r} R$ consists of an AGS-morphism r such that the algebras L_{DS} and R_{DS} are term algebras, and the component r^{DS} is an identity. A match $m : L \to G$ is a total AGS-morphism.*

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "m*"] \\
G \arrow[r, "r'"] & H
\end{tikzcd}$$

Figure 2.1: Attributed graph transformation

Definition 2.4 (Attributed graph transformation) *Given a rule $p : L \xrightarrow{r} R$ and a match $m : L \to G$, an attributed graph transformation $G \overset{p,m}{\Longrightarrow} H$ is defined by the pushout (1) in category $\underline{\underline{AGS}}$, as shown in figure 2.1.*

An attributed graph transformation $G \overset{p,m}{\Longrightarrow} H$ can be constructed component-wise by transforming the graph and the data parts of the AGS-Algebra G separately. (compare figure 2.2):

1. Construct the graph transformation $G_{GS} \overset{p,m}{\Longrightarrow} H_{GS}$ as a pushout in category $\underline{\underline{GS}}$, compare [Löw93].
2. Construct the graph transformation $G_{DS} \overset{p,m}{\Longrightarrow} H_{DS}$ as a pushout in category $\underline{\underline{DS}}$, compare [Bur86].
3. Add the attribute assignments as unique extensions of the resulting diagrams, compare [LKW93].

$$
\begin{tikzcd}
L_{GS} \arrow[rr, "r_{GS}"] \arrow[dr] \arrow[dd, "m_{GS}"'] & & R_{GS} \arrow[dr] \arrow[dd] & \\
& L_{DS} \arrow[rr, "r_{DS}"] \arrow[dd, "m_{DS}"] & & R_{DS} \arrow[dd] \\
G_{GS} \arrow[rr] \arrow[dr] & & H_{GS} \arrow[dr] & \\
& G_{DS} \arrow[rr] & & H_{DS}
\end{tikzcd}
$$

Figure 2.2: Component-wise transformation

2.1.3 Example signature

Here, an example is being presented that illustrates the theoretical approach and hints to some peculiarities.

Example 2.1 (Vectors with two components) *AGS_v is a signature according to the Definition 2.1. It represents vectors with two components and an adding operation.*

$$
\begin{aligned}
GS_v \;=\; & \underline{Sorts}\ Vec,\ Add,\ Vec_{attr} \\
& \underline{Opns}\ src_{Add}, tar_{Add} : Add \to Vec \\
& \phantom{\underline{Opns}}\ vecattr : Vec_{attr} \to Vec
\end{aligned}
$$

$$
\begin{aligned}
DS_v \;=\; & NAT\ + \\
& \underline{Sorts}\ Pair \\
& \underline{Opns}\ select_1, select_2 : Pair \to Nat \\
& \phantom{\underline{Opns}}\ replace_1, replace_2 : Pair, Nat \to Pair
\end{aligned}
$$

NAT the signature of natural numbers.

$$ATTROP_v = \quad attr_{elems} : Vec_{attr} \to Pair$$

The figure 2.3 illustrates the signature. The sorts are shown in frames, all unary operations are shown by dashed arrows. Vec is a node sort, *Add* is an edge sort and Vec_{attr} is an attribute carrier sort of the graph structure. The graphs to this signature are displayed in a simplified way, compare figure 2.4. Objects of node sorts are put in frames, together with their attributes, and edge sort objects are represented by labeled, uninterrupted arrows.

Figure 2.5 shows a rule with its match and the transformation result according to the definitions 2.3 and 2.4. The dashed arrows represent mappings between graph objects.

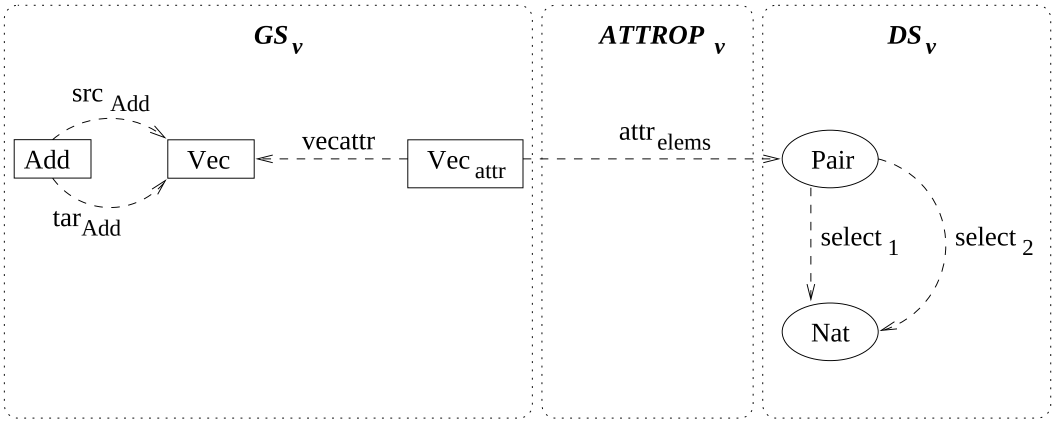

Figure 2.3: Graphical representation of AGS_v from example 2.1

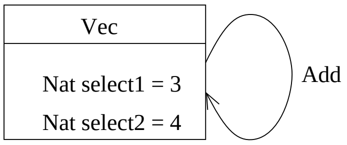

Figure 2.4: Attributed graph structure, described by AGS_v

This rule is non-injective, since it maps more than one object of the left rule side to the same object on its right side. Figure 2.6 displays a non-injective match of the same rule.

The figures 2.5 and 2.6 make clear that, given the presented formal approach, non-injective rules as well as matches are possible. The examples show that this can make sense.

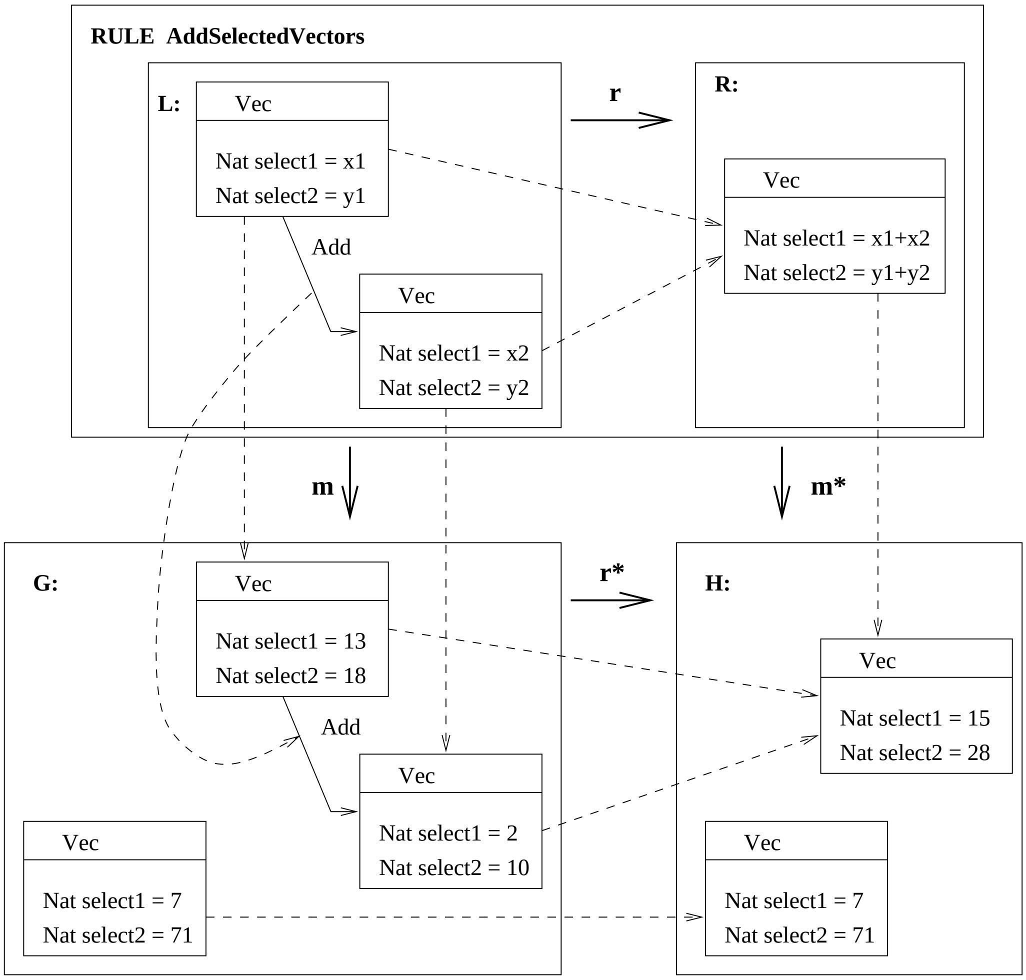

Figure 2.5: Rule transformation for AGS_v

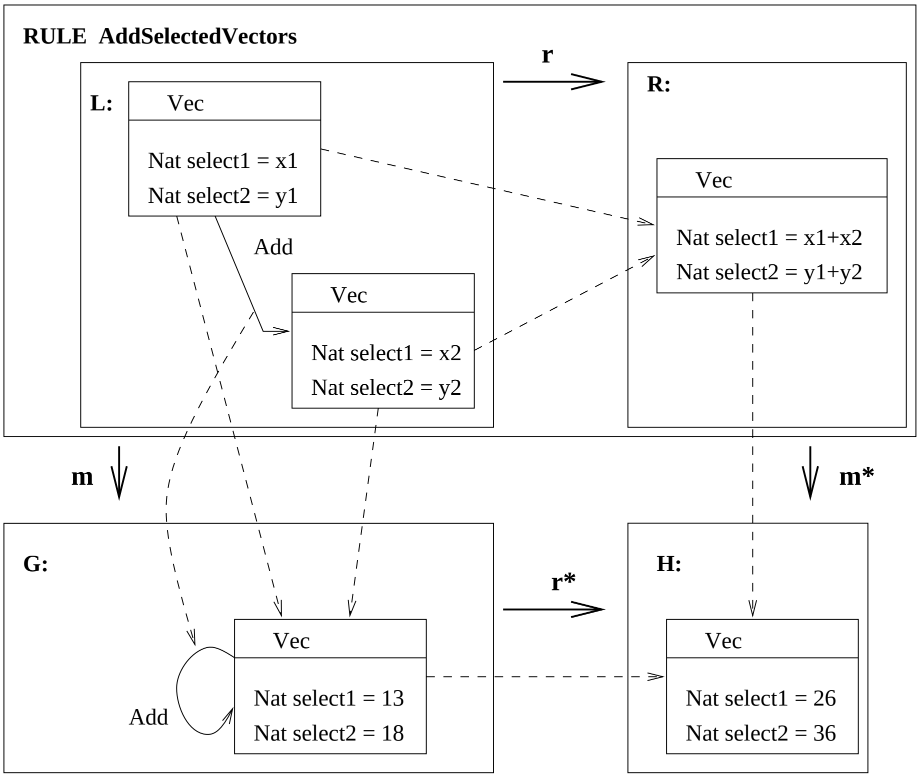

Figure 2.6: Rule transformation with a non-injective match

2.1.4 Concepts of attributes in a tool

In the following sections, realization aspects of theoretical concepts are discussed.

2.1.4.1 Mapping of graph concepts to objects of a tool

As introduced in 2.1, an attributed graph signature consists of three parts:

- A total algebra A_{GS} wrt. the graph structure signature GS; its elements are managed by the graph basis.
- A partial algebra A_{DS} wrt. the data signature DS; the management of its elements is the scope of this thesis.
- A family of functions for mapping between graphical objects and attributes; corresponds to the association between the graph basis objects and attribute component objects.

2.1.4.2 Typed Attribute Tuples

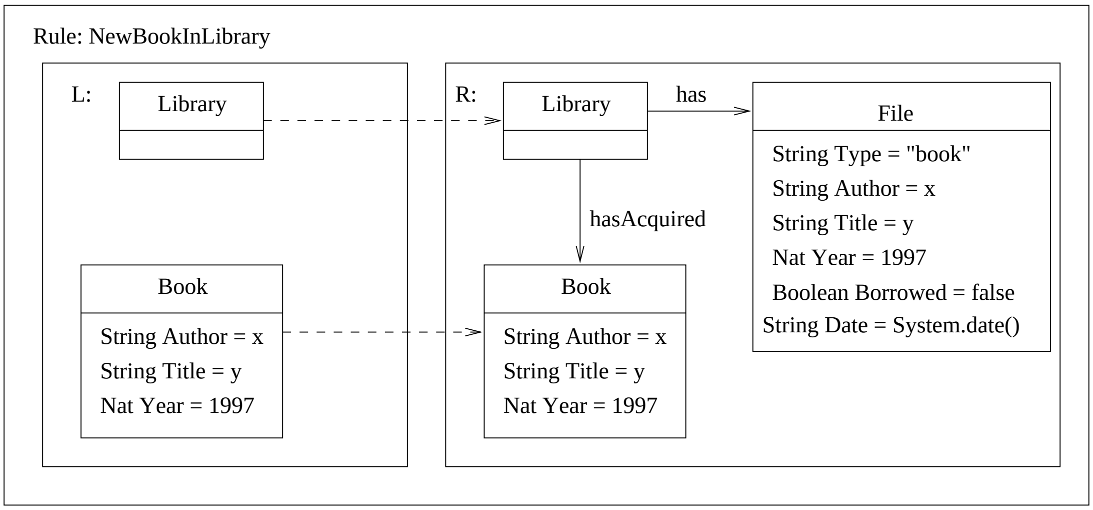

Figure 2.7: Example for a graph rule with attribute tuples

A common use of attributed graphs and rules is shown in figure 2.7[1]. The decision should obviously be to realize an attribute component that maps each graphical object to a tuple of *values*, called *attribute tuple instance*. As shown in example 2.1, this can be expressed by using tuples in the data signature. Generally formulated, the following signature can be established:

[1]Please note that the right rule side can also contain method calls (this example shows a query for the system time).

Example 2.2 (Parameterized tuple signature TS within DS)

$$
\begin{aligned}
DS = \ & \underline{Sorts}\ \ MEMBER_n\ (n \geq 1) \\
& TS = \underline{Sorts}\ TUP_n\ (n \geq 1) \\
& \qquad \underline{Opns}\ select_m : TUP_n \longrightarrow MEMBER_m\ (1 \leq m \leq n) \\
& \qquad \qquad replace_m : TUP_n, MEMBER_m \longrightarrow TUP_n\ (1 \leq m \leq n)
\end{aligned}
$$

In practice, the graph objects should be typed such that each graph object type is mapped to exactly one attribute tuple type. The latter contains zero, one or several *declarations*. One single declaration consists of a *declaration name* and a *declaration type*. This approach fits very well for specification using graph grammars. A short explanation:

Just one attribute entry per node would be a severe restriction. Non-graphical information that belongs together would have to be coded by a complex graph structure. On the other extreme, arbitrary attribute collections would lead to high costs for preserving consistency and when mapping objects. Such an attribute model could not meet the requirements of an efficient specification language.

Since every graph object has exactly *one* attribute tuple instance, from now on the shorter term (typed) *attribute instance* or just *attribute* will be used. Similarly, *attribute type* stands for attribute tuple type.

Correlation between the graph theory terminology and the introduced definitions:

Term	Corresponding theoretical concept
Attribute type	Sort TUP_n in signature TS
Attribute instance	Elements of the algebra A^{TS} wrt. sort TUP_n
Declaration member	(the pair [Declaration name, Declaration type])
Declaration name	Selector $select_n$ in signature DS
Declaration type	Sort $MEMBER_n$ in signature DS
Value member	Elements of algebra A^{DS} wrt. sort $MEMBER_n$

Note: The attribute carrier set is not reflected in the tool design. Instead of deleting the carriers and creating them anew during graph transformation, the respective values in the attribute instances are just *replaced*.

It can be said that the attribute component creates a minimal data signature that allows tuples to be used with sorts for attribution of graph objects. Arbitrary sorts can be added to the signature. On the other hand, the data signature should be powerful enough as to include calculations on natural numbers, character strings or, as already mentioned, evaluation of Java expressions. It therefore seems proper to structure the attribute component as two parts. Figure 2.8 shows the relationship between the envisioned components and the underlying theory.

The graph basis has to keep track of the attribute type for each graph object type it creates. Whenever creating a new attribute instance, it is the responsibility of the graph basis to specify the corresponding attribute type.

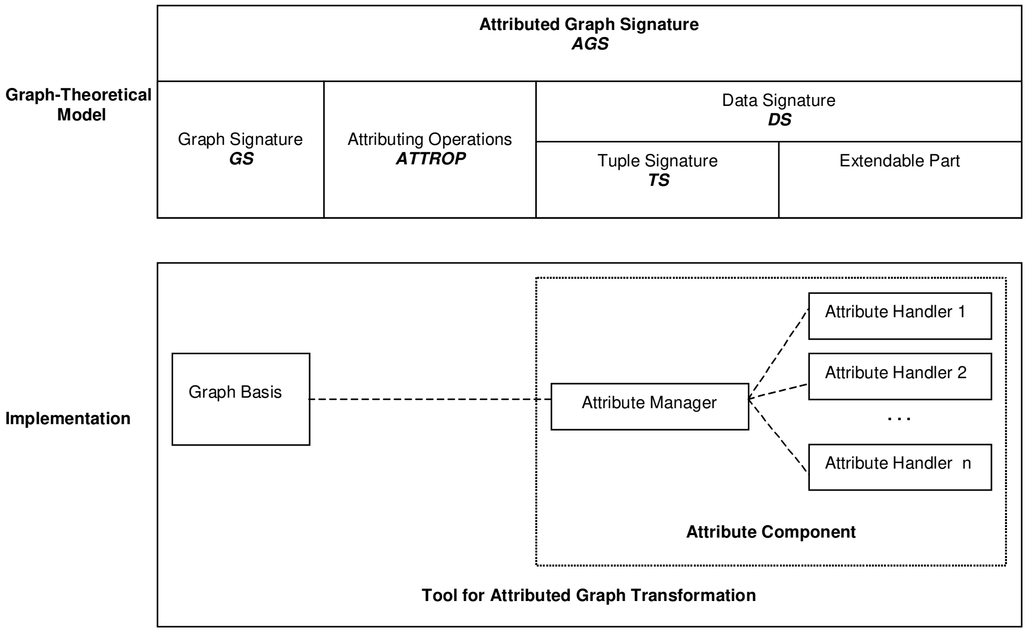

Figure 2.8: Modularization as a reflection of graph theory concepts

2.1.4.3 Allowed Expressions

Now some assumptions and constraints are made wrt. the graph transformation system.

- Variables in objects on the left rule side implicitly create corresponding declarations. Example: if a graph object has an attribute instance with a declaration member `quantity : Integer` and its value set as `quantity =` X, a new variable declaration is created: X `: Integer`.

  On the other side, terms like `quantity =` X `+ 1` are not valid here. [2]

- Variables on the right rule side are refer to those already declared. Non-declared variables may not appear there.

- The host (working) graph can only contain constant values.

The following table sums up all possible values.

	Constant	Single Variable	Complex Expression
Example	42	X	7 * X - 13
Working Graph	Yes	No	No
Left Rule Side	Yes	Yes	No
Right Rule Side	Yes	Yes, if declared	Yes, if all var. declared

[2] An expression of the form $n + 1$ in the left rule side is a kind of an implicit application condition. The implementation costs are kept much lower if the form of application conditions is always explicit, i.e. limited to rule-global Boolean expressions, in this case: $n > 0$.

2.1.4.4 Match

A *match* is a total morphism from a graph (left rule side) to another one (working graph). It means that each graph object and its attribute instance finds a corresponding object in the target graph. The value members of an attribute instance must "match" wrt. their algebra. In the body of real numbers $\mathbb{R}$ it means the value members must be equal. If, however, value members are sets, *inclusion* could be a more appropriate matching operation. It means that more than one mapping of a value member to another can exist.

If the value member on the left side is a variable, the corresponding value member of the working graph is assigned to it[3].

2.1.4.5 Context

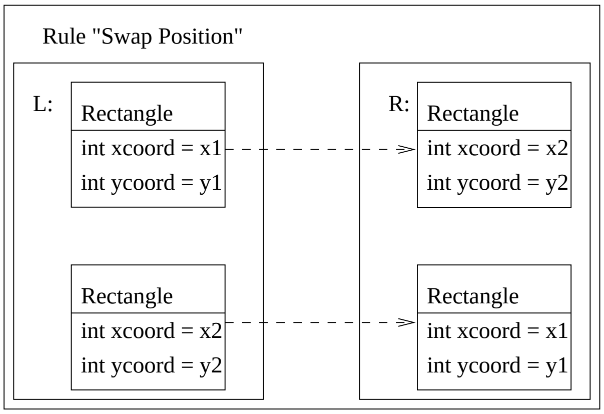

Figure 2.9: Variable access from arbitrary objects within a rule

As figure 2.9 shows, variable assignments created by one graph object can be referred to by expressions within another graph object. Obviously, variables are "rule-global". Therefore, a "context" is required, where all variable declarations of a rule and all assignments of its match are kept.

Since it can be desirable to maintain more than one match of a given rule at a given time, there should be a *rule context* and a *match context*. The rule context contains the variable declarations. A match context always belongs to the context of the rule that the match was created for. It has access to the declarations of its rule context, but the variable assignments it makes are kept local, thus not affecting the rule context.

If this mechanism is applied, not merely in these two steps (rule context $\rightarrow$ match context) but one step further, then negative application conditions in the graph part can also be supported: (rule context $\rightarrow$ match context $\rightarrow$ negative match context).

[3]All value members in the working graph are constants, compare section 2.1.4.3.

Explanation: During a match search with negative application conditions, a match for the positive part of the left rule side is looked for first. If this is found, a match of the negative part is searched for, *using the same variable assignments*. Only if this then fails, the complete match has been found. Apparently, two match contexts are employed, and the second one must "know" the variable assignments made by the first. In practice, this implies four kinds of context:

1. (positive) rule context, contains variable declarations and application conditions;
2. match context for the positive part of the left rule side; stores variable assignments in compliance with the declarations of the rule context;
3. context for the *negative* part of the left rule side, contains attribute application conditions and variable assignments *additionally* to those made by the "positive match context". Additional variables can also be declared in the negative context. Variables declared *only* in the negative context cannot be employed in expressions on the right rule side;
4. Match context for the negative part of the rule; "knows" all the variable values that were previously assigned in the positive match context and stores values for variables added within the negative rule context.

For more about graph grammars with negative application conditions see [HHT96].

2.1.5 Transformation

The attribute transformation will be done by *in-place-replacement*[4].

The graph transformation can be automated, i.e. matches are computed and transformation applied until termination. All possible conflicts should therefore already be prevented at match tests. An error that occurs during transformation and leads to a stop in the derivation chain is not acceptable. That is why all variables referred by the right rule side must be assigned values when matching is performed.

However, there's a conflict case that cannot be prevented by merely testing the declarations and assignments of variables. It has to do with the decision not to model attribute carriers (compare the note on page 19). Consider the non-injective match of the rule in figure 2.10. It is not clear, which of the values (1 or 2) the attribute value of `Radius` should be assigned. The solution to this problem is to perform a special check while matching. It should be checked, whether two or more graph objects from the left rule side are mapped to the same object in the working graph. If this is the case, it is checked whether the

[4] If desired, diagram transformation can be constructed by previously copying the graph G to H and an in-place-replacement in H.

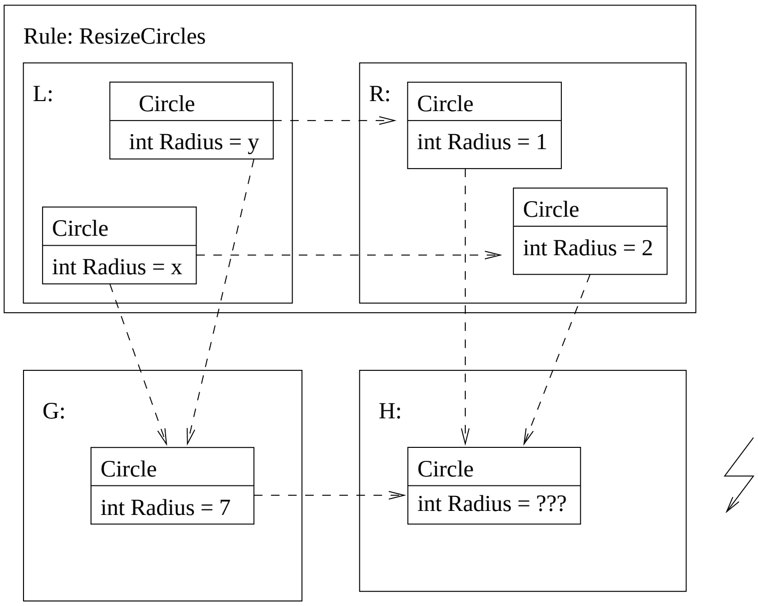

Figure 2.10: Conflict when transforming with a non-injective match

respective value members on the right rule side are equal. If they are not, the match is forbidden.

In connection with the Java expression handler, where arbitrary user-implemented classes can be involved, a transformation success cannot be guaranteed. In case where expression evaluation fails, the Java exception is propagated to the graph transformation unit that invoked the attribute transformation. This case is treated in slightly more detail in section 5.5.

2.2 Application Conditions

Upon presenting the formal concepts of application conditions, an appropriate tool realization approach is discussed.

2.2.1 Formal Definitions

Just as attributed graphs are split in a graphical and an attribute part, application conditions for attributed rules are split as well. There are application conditions concerned with the graphical part that are expressed by GS-homomorphisms of the graph part. The attribute application conditions are expressed by equations that have to be fulfilled under the variable assignments performed by a match.

Definition 2.5 (Equation for SIG) *An equation wrt. to a signature $SIG = (S, OP)$ is a pair $(X, l = r)$, where $X = (X_s)_{s \in S}$ is a family of sets of variables and l, r are terms with variables of X of the same sort.*

The term algebra for SIG and X is denoted by $T_{SIG}(X)$. Satisfaction of equation is introduced as follows:

Definition 2.6 (Satisfaction of equations under evaluation) *Let SIG be a signature, X a set of variables and $p : T_{SIG}(X) \to A$ a SIG-homomorphism, also called* evaluation of terms in A. *Furthermore, let $e = (X, l = r)$ be an equation w.r.t. SIG. Then A* satisfies *e under p, denoted by $A_p \models e$, if $p(l) = p(r)$. The algebra A satisfies a set of equations E under p, denoted by $A_p \models E$, if $A_p \models e$ for all $e \in E$.*

2.2.2 Realization Approach

Since rule conditions are a property of rules, attribute conditions (equations) will be attached to the rule context, like the variables. Integrating equations into the rule context means additional checking during rule matching. Each time a variable is assigned a value, every equation containing this variable is tested. If one of these equations doesn't hold with this new value, the match that involved this value assignment is forbidden. If the equation cannot be evaluated because one or more other variables it contains are still free, the match is accepted and the variable assignment performed.

Figure 2.11 shows a rule with a negative application condition (NAC), where the contexts, consisting of variables and conditions, are also displayed. The rule diagram describes the action of a client borrowing a book from a library. The positive part of the left rule side, L, ensures that the client does not exceed the limit of ten borrowed books at the same time. The negative part of the left rule side, L_{neg1}, ensures that there is not another client (therefore the attribute condition requires that the names not be equal) who has reserved the same book.

The main attribute context, displayed at the right side of the rule box bottom, is "known" within the negative application condition. The attribute context displayed in the NAC box however is local and only accessible in matches of the NAC.

The scope of attribute contexts is illustrated in figure 2.12. The source of an arrow "knows" about the variables and conditions in the arrow target.

There can be zero, one or more NACs for a rule. In order to find a match for a rule, the positive left rule side L has to match the working graph, while under the resulting context (variable assignments), none of the NACs matches the working graph.

This means that for every NAC of a rule, there is a NAC rule context, and for every NAC match, there is a NAC match context. In other words, a rule with n NACs has n NAC rule contexts and, when looking for a match, n NAC match contexts.

The following table expands the table for possible expressions on page 21 by including negative application condition (NAC) contexts. Note that a rule NAC context allows the same attribute values as a left rule side context.

	Constant	Single Variable	Complex Expression
Example	42	X	7 * X - 13
Working Graph	Yes	No	No
Left Rule Side	Yes	Yes	No
Rule NACs	Yes	Yes	No
Right Rule Side	Yes	Yes, if declared	Yes, if all var. declared

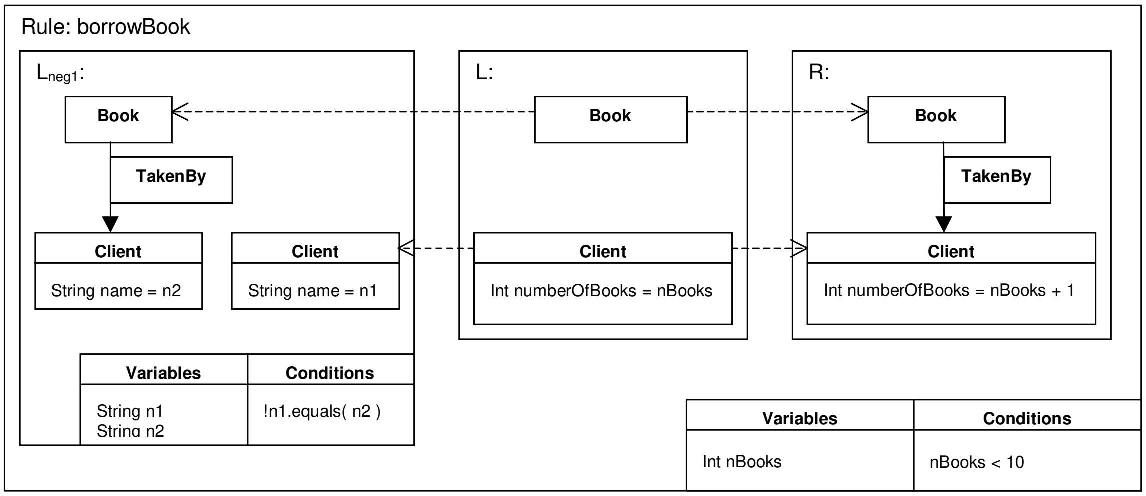

Figure 2.11: Example for a rule with a negative application condition (NAC)

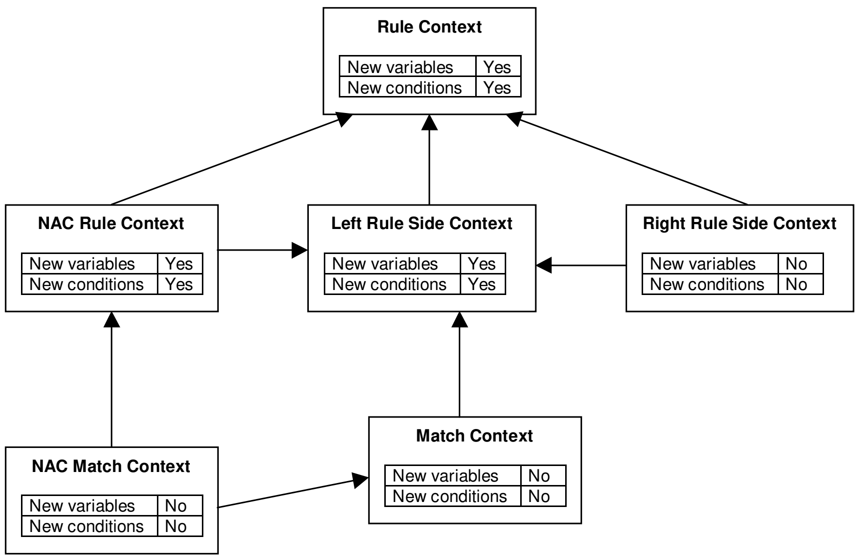

Figure 2.12: Scope (visiblity, access) of attribute contexts in graph transformation

2.3 Distribution Concepts

In this section, the concepts of distributing attributed graph transformation over a network are described, an example is given and the realization approach is discussed. The concepts and definitions follow [TFKV98].

2.3.1 Formal Definitions

A network of attributed graphs is defined by a network graph and a diagram of attributed graphs and graph morphisms, where the diagram structure is induced by the network graph. This is formalized by a functor from the network graph into a suitable category. This category, called ALG_{ags}, contains all AGS-algebras and -homomorphisms for all attributed graph signatures, if we allow different attributed graph signatures for different local graphs.

A distributed transformation step on attributed graphs is characterized by a single-pushout in the category $DISTR(AGRAPH)$ of networks of attributed graphs and morphisms in between.

Definition 2.7 (Category $DISTR(AGRAPH)$) *The category $DISTR(AGRAPH)$ of distributed attributed graphs has the following objects and morphisms:*

- *Given an attributed graph signature Σ, an object of $DISTR(AGRAPH)$, called network of attributed graphs, is a pair $\hat{A} = \langle A, A' \rangle$ where A is an AGS-Algebra w.r.t. Σ, and A' is a functor between the category induced by the graph part A^G of A and the category of all AGS-algebras for all attribute graph signatures ALG_{ags}.*
- *An arrow $\hat{f} = \langle f, f' \rangle : \langle A, A' \rangle \to \langle B, B' \rangle$, called NAG-morphism, is a pair with $f : A \to B \in \Sigma - AGS$ and $f' = (f'_i)_{i \in Fcat(f_G)}$ is a natural transformation between the two functors A' and $B' \circ Fcat(A_G)$.*
- *If f is injective (surjective), $\hat{f}$ is called n-injective (n-surjective). If moreover, all $\hat{f}'_i$ with $i \in Fcat(A_G)$ are injective, $\hat{f}$ is called injective, too.*

Definition 2.8 (Network of attributed rules and matches) *A network of attributed rules $\hat{p} : \hat{L} \xrightarrow{\hat{r}} \hat{R}$ consists of two NAG-morphisms $\hat{l}$ and $\hat{r}$ such that all algebras in L^D and R^D are term algebras and r'^D is identity. A network of attributed matches $\hat{m} : \hat{L} \to \hat{G}$ is an NAG-morphism such that for all $i \in |FCat(L^G)| : \Sigma_i = \Sigma_{m(i)}$.*

Definition 2.9 (Network of attributed graph transformation) *Given a rule $\hat{p}$: $\hat{L} \xrightarrow{\hat{r}} \hat{R}$ and a match $\hat{m} : \hat{L} \to \hat{G}$, a network of attributed graph transformations $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ is defined by a pushout in category $DISTR(AGRAPH)$ as shown in figure* 2.13.

$$\begin{array}{ccc} \hat{L} & \xrightarrow{\hat{r}} & \hat{R} \\ \hat{m} \downarrow & & \downarrow \hat{q} \\ \hat{G} & \xrightarrow{\hat{r}'} & \hat{H} \end{array}$$

Figure 2.13: Single-Pushout in category DISTR(AGRAPH)

2.3.2 Example

An example for distributed graph transformation shall clarify the needed concepts and motivate the realization approach. Let us consider the following scenario:

A school has this amazing service, where for each course that a student signs, the book for that course is reserved for him at the local city library. This yields two local systems, the school and the library. Consider the figure 2.14. It displays the types of interface types and local types involved. *Person* is a type interface with local types in the library and in the school system. *Book* is a type interface that also has corresponding local types in both the local systems. In *Library,* the type *Person* is employed to describe the clients, while in *School,* it describes the students. Note that the local systems are free to extend the types locally. Thus, the *Library* system adds the *Book* attribute *isAvailable* and the *Person* attribute *numberOfBooks,* while the *School* system adds the *Person* attribute *faculty.*

Now look at figure 2.15. The "left rule side" is on top of the "right rule side". If a student takes a course, the book that the course follows is reserved for him at the library. The local condition on the school side is that the course is within the faculty at which the student pursues his studies. The conditions on the library side are that the student (client) has not reached the limit of ten borrowed books. Note: The local variables and conditions are accessible within each respective local system on both rule sides. It is also possible to have a "global", i.e. a context interface, in which case its variables and conditions are accessible within each of its local contexts. The same principle holds for contexts of negative application conditions NACs.

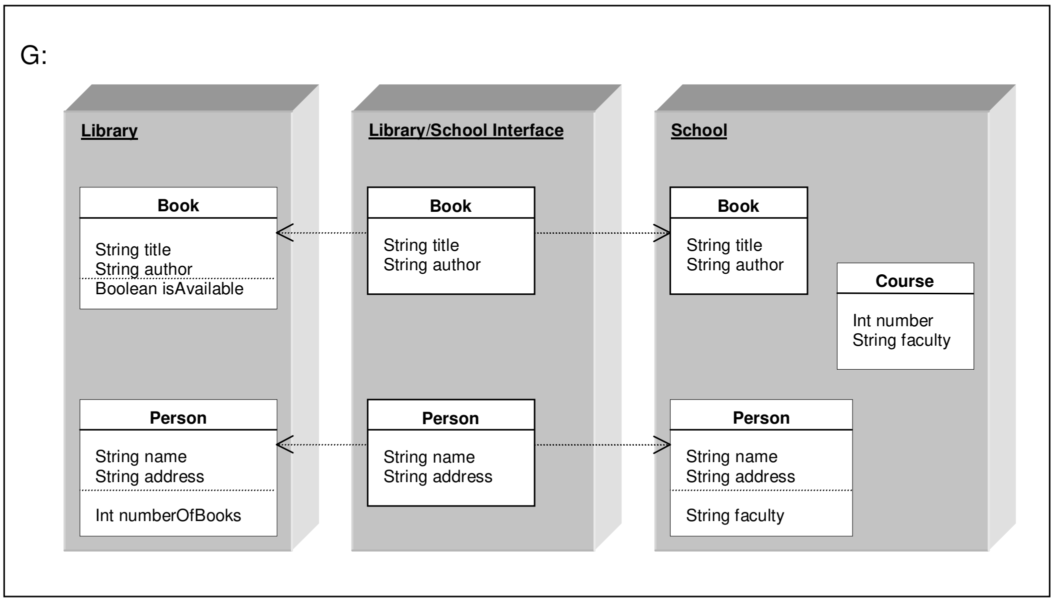

Figure 2.14: Attribute types in a distributed system

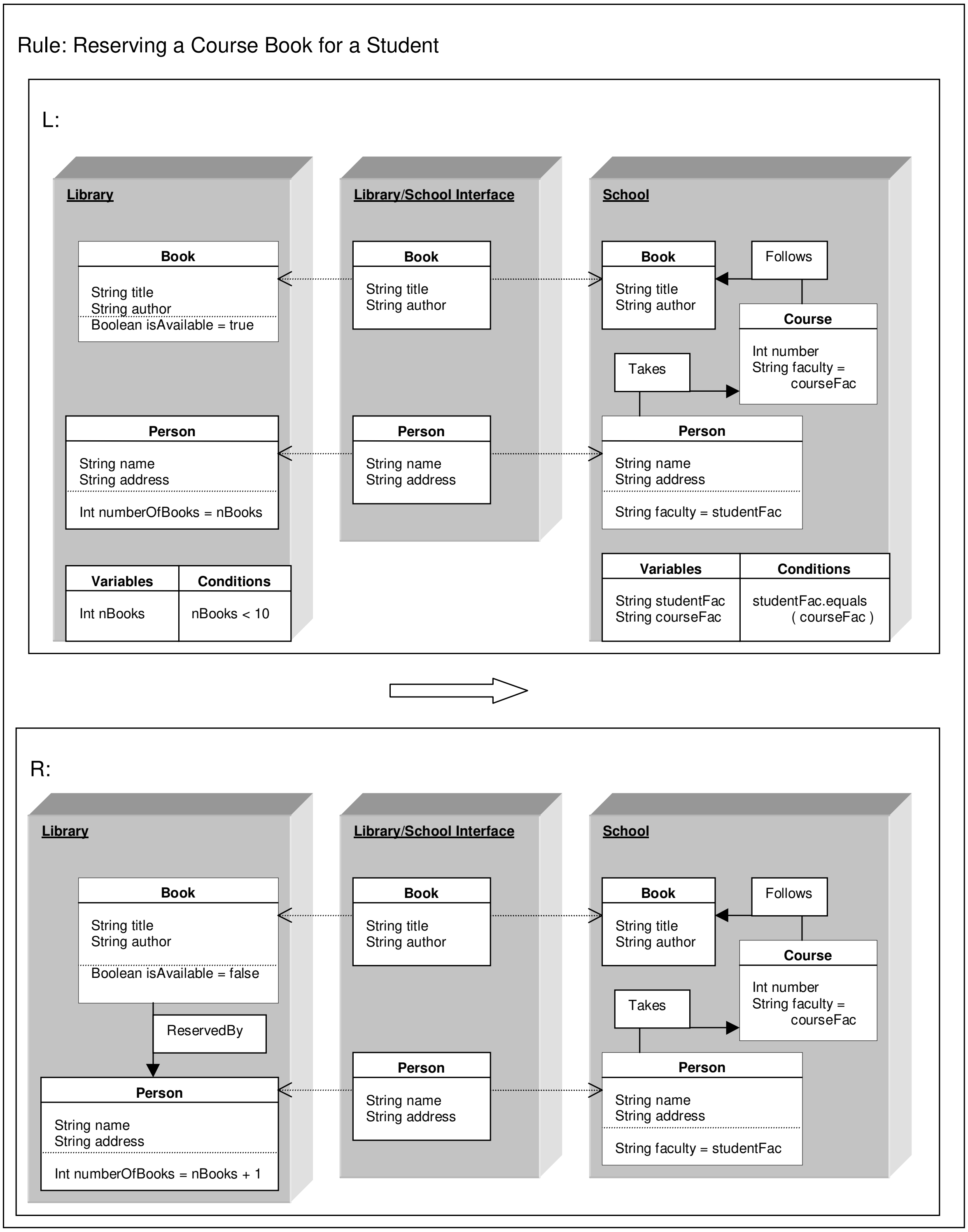

Figure 2.15: A distributed rule

2.3.3 Realization Approach

From the previous section, the following approach is derived:

The attribute component should allow constructing interface-relationships between attribute objects:

- If an attribute type T_{int} is the interface of another attribute type T_{local}, then all the declaration members of T_{int} must also be in T_{local},
- If an attribute instance I_{int} is the interface of another attribute instance I_{local}, then the corresponding value members must always be equal.
- If a rule context C_{int} is the interface for another rule context C_{local}, then:
  - All variable declarations of C_{int} must also be in C_{local};
  - The values of variables which are also declared in C_{int} must always be equal.
  - All conditions of C_{int} must also be in C_{local};

One attribute object can be the interface of zero, one or more other attribute objects, and one attribute object can be connected to zero, one or more interface objects.

2.4 Accepted (Java-) Expressions for Attribute Values

The first, exemplary attribute handler should already be able to interpret expressions. Their syntax is established as an appropriate subset of the Java language syntax.

Java syntax borrows a lot from C. There are:

- "primitive" types[5]:

  `byte, char, short, int, long, float, double, boolean`

- arithmetic expressions with operators:

  `+, -, *, /, %`

- Boolean expressions using

  `!, ==, !=, <, >, <=, >=, ||, &&`

[5]In Java, *char* is two bytes long and complies with the Unicode standard

- bit manipulations:

  ```
  ~, |, &, ^
  ```

- array declarations:

  ```
  new int[4][][]
  ```

- array referencing:

  ```
  anArray[3][0]
  ```

- *conditional expressions* with the `?` - Operator:

  ```
  <Condition> ? <Then-Expression> : <Else-Expression>
  ```

Strings of characters are *not* "primitive" types, but instances of the Java standard class *java.lang.String*. Handling of classes and objects goes as follows:

- Creating a new instance by calling a constructor of a class, e.g.:

  ```
  new Random()
  ```

- Access to variables of classes and instances:[6]

  ```
  java.lang.System.out;  aPoint.x;
  ```

- Invocation of class methods (those declared as `static`):[7]

  ```
  java.lang.Math.sin( 0.5 )
  ```

- Invocation of instance methods:

  ```
  "abcdefg".length();  aVeryLongString.length();
  howDoesThisLook.toString();
  ```

  Here, `"abcdefg"` is a constant String instance and `howDoesThisLook` is a variable of an arbitrary class. In Java, the class Object is the root of any class inheritance hierarchy, i.e. every class is a subclass of Object. The method `toString()` is defined in Object and thus can be invoked for any class.

Any Java class can be used in the expressions, be it a standard Java library class from *(JDK, Java Developers Kit)* or self-defined one. It should be noted that invoking methods for arbitrary objects can endanger the consistency or determinism of graph transformation, since methods can have side effects (or create random numbers).

[6]In the actual implementation, class paths must be enclosed in $, see page 82. For this example, the first term should then read `$java.lang.System$.out;`

[7]In the actual implementation, class paths must be enclosed in $, see page 82. For this example, the first term should then read `$java.lang.Math$.sin( 0.5 )`

Chapter 3

Applied Modeling and Implementation Tools and Methods

This chapter introduces the employed tools and gives reasons why they are chosen. First, it presents the modeling notation UML, followed by the applied design patterns. It then goes on to describe the implementation language Java. Finally, the employed parser-generating tool JavaCC is discussed.

3.1 Unified Modeling Language (UML)

One of the first modeling standards for the object-oriented method (OOM) was the OMT model, introduced by J. Rumbaugh [RBP$^+$91]. It offered a strong support for the description of the static aspects of a system. At the same time, the design technique by G. Booch [Boo91] became popular, mainly due to its expressiveness when modeling dynamic system features. Apart from these, there is a multitude of other techniques, and each of them has a somewhat different notation.

Meanwhile, the relatively new field of *Design Patterns* is growing [GHJV95]. Design patterns that are repetitively used in software projects are developed, described and standardized. This calls for a common modeling notation.

With this in mind, the teams of Rumbaugh and Booch came to work together. The result is the design of the *Unified Modeling Language* (*UML*). This document is a recommendation that suggests various notational possibilities. It is addressed to software developers, but primarily it is a notation specification for software development tools.

It offers little support for the modeling itself, i.e. the methodology. However, the advanced notation comprises many powerful concepts. Moreover, a trend towards *UML* is evident, and therefore its notation was adopted by this thesis.

UML contains a great variety of notational concepts. Static, dynamic, distribution, reactive, real-time and other aspects are included. Here, only the subset that is employed in this thesis is described.

3.1.1 Static Diagrams

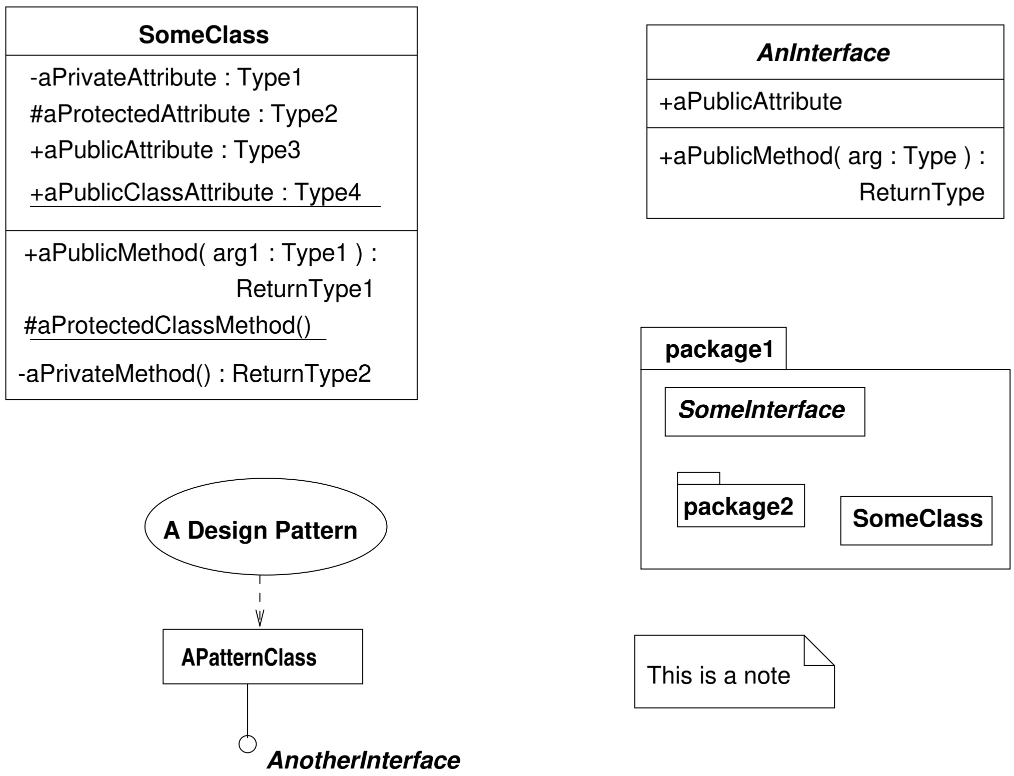

Figure 3.1: Static elements of the UML-notation

The figure 3.1 shows the static elements that are employed in this thesis. The central modeling structure of the OO-method is the *class*. It can contain instance and class variables and methods. These can be *public*, *protected* or *private*, and these properties are expressed by the *visibility marker* $+$, $\#$ or $-$, respectively. Methods can have parameters and they can return values. The declaration syntax follows that of *Pascal* or *Ada*. Class methods and variables are underlined.

Variables and/or methods can be omitted in order to improve clarity in diagrams. When neither methods nor variables are shown, a simple rectangle box around the class name is enough.

Abstract classes are termed interfaces and are distinguished from concrete (implementing) classes by setting their names in Italics font. They can also be displayed as a little circle connected to their implementing class box. This means that the concrete class implements *at least* the interface declarations, but possibly more.

Another structure element is the *package*[1]. A package can contain classes, interfaces and again other packages. Above all, a package defines a naming space, such that SomeClass in the figure above can be addressed unequivocally as package1::SomeClass. Apart from that, there are language-specific meanings. Thus, in Java, objects of the same package have privileged access rights for some variables and methods of their "neighbor classes".

In UML, package names begin with a capital letter, but not in Java. This is one of the notational differences between UML and Java. The following table shows various such deviations and gives the notation adopted within this thesis:

	UML	Java	Thesis
Package	CapitalLetter	smallLetter	smallLetter
Path	Package::Class	package.Class	package::Class
Declaration of methods and variables	`var : Type` (as in Pascal)	`Type var` (as in C)	`var : Type` (as in UML)

The principle being followed is to preserve the UML notation as far as possible, except where this would lead to a name change of the implemented components, as in the first example.

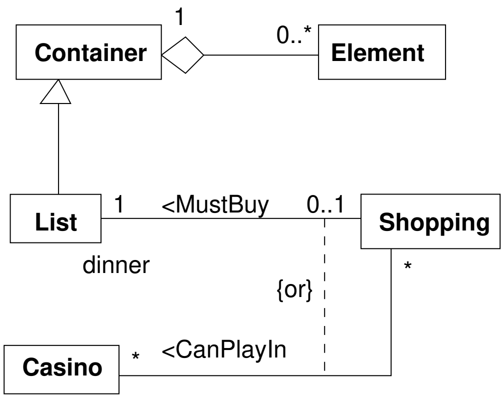

Figure 3.2: Associations and Generalizations

The figure 3.2 displays notation of associations and abstractions. In this example, List is a specialization (subclass) of Container, which can contain zero to many instances of Element. An association can have a label (`MustBuy`). An arrow ("<") can show, in which direction the describing label should be read. However, the label has no concrete semantics, in particular it is not an access variable. Contrary to that, the role descriptor (`dinner`) expresses that the class Shopping has a variable `dinner` of type List. The interrupted line with the {or}-label describes a constraint on the associations that it connects. An instance

[1]This is not to be confused with the *package*-concept of the language *Ada*. There, it denotes a certain kind of class.

of a class that is the source of such a "fork" can have at most one of these associations at a given time.

3.1.2 Dynamic Structures

Object interactions are expressed by sequence diagrams. The figure 3.3 shows the typical cases.

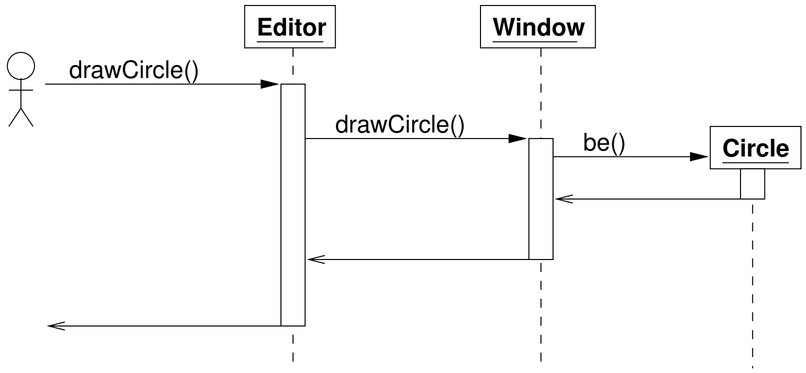

Figure 3.3: Example of a sequence diagram

The time axis goes from top to bottom. The interrupted line represents the object life span. An object is an instance of the class whose name is displayed and underlined in the respective box. The high slim rectangles along the time lines indicate activation times, i.e. the time span of a method execution.

In the example, the human user activates the already existing Editor object which activates the Window object. The latter creates a new, previously non-existing Circle object.

Moreover, there are elements expressing conditional execution, loops, object destruction and recursion. For further details please see *UML Notation Guide* ([Rat97]).

3.2 Design Patterns

Design patterns describe solutions for problems in software development that occur over and over again. This section does not intend to provide a complete introduction to the subject. Rather, it lists and briefly describes the design patterns used in this thesis and explains why and where the pattern is used. For a detailed description of the pattern itself, the reader is referred to [GHJV95], which is an excellent work and meanwhile a classic on the subject. The terminology introduced there is used throughout the system design presentation of this thesis.

Abstract Factory "Provides an interface for creating families of related or dependent objects without specifying their concrete classes." [GHJV95] (p. 87ff)

The attribute component consists of an interface part and an implementation part. The interfaces are used by the client (e.g. a graph transformation system) and are not subject to frequent change. The implementation part is therefore flexible and can be changed more freely (or even replaced), as long as it implements the interface. This makes the existence of an *Abstract Factory* necessary for creating attribute entities without specifying their concrete implementation classes.

Facade "Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a *facade* object that provides a single, simplified interface to the more general facilities of a subsystem." [GHJV95] (p. 185ff)

The attribute component contains a multitude of interfaces, and most of them have methods for various purposes. For clients that are interested in only one particular system aspect (like information access), a facade interface is provided that contains only this aspect's services. The attribute component has three facade interfaces:

- attribute information access;
- mapping and transformation of attributes;
- calling and control of attribute editors.

Proxy "Provides a surrogate or placeholder for another object to control access to it." [GHJV95] (p. 207ff)

A rule context in the attribute component is shared by all attributes in a rule. However, the attribute expressions on the left side of the rule have different restrictions than those on its right side (see p. 21). Using two different context proxy objects (one for the left and one for the right side of the rule), all variables and conditions of a rule context can be shared, while at the same time restricting the attribute expressions in an elegant way.

Interpreter "Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language." [GHJV95] (p. 243ff)

This pattern typically uses an inheritance hierarchy where each non-terminal is represented by a class. This allows reusing code for type checking and evaluating of expressions.

The Java expression handler implemented within the attribute component uses an interpreter which is generated by the tool JavaCC[2]. It also uses a hierarchy of classes

[2]See section on JavaCC in this chapter.

that represent non-terminals and terminals and implement their type checking and evaluating methods.

Mediator "Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently." [GHJV95] (p. 273ff)

In the attribute component, the Mediator pattern is employed in the view mechanism. The information about the order and hiding of members in an attribute tuple is kept separate from the tuple itself, since it is only relevant for displaying purposes. But instead of having the client keep a separate view entity for each tuple, it is enough to operate with just one view object per scenario, which has internal access to format information for each tuple.

Observer "Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [GHJV95] (p. 293ff)

This pattern is also known as Model-View-Control pattern in Smalltalk and Observer/Observable in Java.

In the attribute component, this pattern is used rather extensively. An attribute tuple type is dependent upon its super type, a tuple instance is dependent upon its type, an attribute editor upon the displayed tuple etc.

3.3 Java

Java is a programming language developed by *Sun Microsystems* and it is getting more and more important. This is due to the following characteristics that make this language attractive for many applications:

- object oriented;
- compiles the source code into a machine independent "byte code" that runs on every platform;
- so-called *applets* (portable programs) can be sent/downloaded via a computer network and are immediately executable on the recipient machine;
- the Java syntax is C++ oriented, but Java is less complex:
  - no pointers, resulting in fewer error opportunities;
  - no preprocessor;
  - no separate header files (the declarative information is automatically generated by the Java compiler);

  - no operator overloading;
  - no multiple inheritance (only single inheritance);
  - automatic garbage collection.

- already contains extensive standard libraries, including
- *Abstract Window Toolkit (AWT)*, a widget library.

The following is a list of reasons why Java was selected for this particular thesis:

- The programs are executable everywhere, therefore very suitable for an experimental use in an academic framework, with students being able to run the system on their PC at home;
- Java has a stronger object orientation than C++, this leads to cleaner designs;
- the concept of *interfaces* enhance greater flexibility;
- the *Reflection Package* (java.lang.reflection) offers the possibility to find classes dynamically during runtime (using their name string) and to access their methods and variables. This makes the implemented expression interpreter very flexible and easily extendible;
- the existence of a ready parser generator *Java Compiler-Compiler (JavaCC)*, a *Sun Microsystems* product free of charge, accelerates the development of an interpreter;
- the possibility to send objects within a computer network facilitates the development of a tool for *distributed* attributed graph transformation at a later project stage;
- the built-in mechanism (the java.io.Serializable interface) offers semi-automatic serialization of arbitrary objects and thereby facilitates writing and reading of structures where persistence is required.
- the garbage collection, the window toolkit and the other libraries help saving development time.

Java is shipped as JDK (Java Development Kit). This contains, among others, *javadoc*, a program for automatically generating HTML (Hypertext Meta Language) documentation from source code. Together with further conversion software (HTML → Latex), this tool was applied in order to extract the documentation presented in the appendix of this thesis from the source code.

The system was implemented using the Java version JDK-1.1.5. Current documentation can be found under [Sun97b]. As Java documentation, [Fla96] was also used during system development. However, it describes the outdated version JDK-1.0, so that a newer edition is strongly recommended.

3.4 Java Compiler-Compiler (JavaCC)

The *Java Compiler-Compiler* already mentioned in a previous section processes YACC-like input and generates the corresponding parser for a given grammar. At the same time, a scanner for lexical analysis is created, whereas in a YACC framework the separate tool LEX is employed for this task. In the case of a syntax error, the generated parser gives a simple error description with the cursor line and column position and the set of expected tokens at cursor.

For each production, a piece of Java code can be provided. An included preprocessor for JavaCC, *JJTree*, proved to be especially useful for the Java expression interpreter implemented in this thesis. JJTree facilitates constructing of an abstract syntax tree during parsing. Special keywords (macros) control where a new node is added and for which production classes.

On a practical level, this means that the *Interpreter* design pattern ([GHJV95], p.243ff), including object-oriented compiler design, is already integrated. There is the interface *Node* that has to be implemented by each node class of the abstract syntax tree. This separates execution code from the grammar and thereby increases the clarity, extensibility and reusability of a compiler specification. The inheritance hierarchy helps structuring and saving code. In an included example for evaluating of arithmetical expressions, each node class is a subclass of the root node class SimpleNode. The node class specific functionality is created by simply overriding its `interpret()` method.

The Java expression handler was implemented using JavaCC version 0.6(Beta) and JJTree version 0.2.4. Documentation for JavaCC and JJTree can be found in [Sun97a].

Chapter 4

Requirements Specification

After the graph transformation concepts in chapter 2, the required functionality and structure of the attribute component are briefly presented here. They are derived from the concepts described in the chapter 2 as well as from considerations about the convenience of usage and the extendibility of a system. Sometimes, the appropriate design pattern is also suggested. For a brief description of design patterns, see section 3.2.

Here is the list of the main features:

- typed attributes, as tuples (sets) of actual attribute values;

  As shown in section 2.1.4.2, the most recommendable way to organize attributing information for graphical entities is by grouping it into tuples. Each instance tuple has a type tuple whose members describe the types and selectors (names) of the corresponding instance members. The instance members contain the actual attribute values or expressions.

- the possibility of adding new value types to the system;

  To warrant extensibility of the attribute system with respect to the set of possible attribute types and values that can be assigned to a graphical entity, it is a requirement that new value types can be added to the system easily. Thus, the attribute component is divided into the attribute manager that processes the attribute tuples and a set of attribute handlers providing functionality for attribute member values. This division also reflects the theoretical approach described in section 2.1.4.1.

- an attribute handler for processing of Java expressions;

  The first, included attribute handler should be able to process common Java expressions as described in section 2.4. Thus, not only basic types like Integer or String, but also arbitrary Java classes, can be employed within the attribute component.

- a transformation rule concept, with:

- mapping from attributes on the left rule side to those on the right rule side;

  A graph transformation rule has a left side and a right side. Objects on the left side can be mapped to objects on the right side, causing these objects to be preserved by the rule application, even though the attribute values may change. In order to construct transformation rules, the graph component allows to map a left rule side object to a right side one. Accordingly, the attribute component must offer a service to map an attribute tuple to another. This information enables the attribute component to check for inconsistencies and characteristics of a rule. For example, it can be checked if a rule is injective.

- variables and parameters;

  A variable usually appears as the expression of an attribute on the left rule side. It is set when this attribute is matched to an attribute in the working graph. Apart from that, a variable within a rule can be assigned an expression, possibly containing other variables. As such, it assumes the role of an internal function and can be used to avoid unnecessary multiple occurences of terms in expressions on the right rule side. A combination of such variables (functions) can be used to perform intermediate calculations. A special case of such a function is a constant.

  A parameter can either be an input parameter, an output parameter or both. An input parameter is a variable that is given its value interactively either before looking for a match (when its value constitutes an application condition) or before rule application. The expression assigned to an output parameter is evaluated after matching its rule to the working graph, immediately before rule application. All variables can be employed (referenced) in attribute expressions on the right rule side.

- application conditions;

  An application condition is an expression of the type Boolean. All application conditions have to be fulfilled in order for a match to be successful.

* a rule-match concept, with[1]:

  - mapping from attributes on the left rule side to those in the host graph;
  - assigning concrete values to the variables declared in attributes on the left rule side, if need be;
  - checking the rule application conditions;

The realization approach for keeping variables and conditions in a rule-global context is described in section 2.1.4.5. It includes four kinds of contexts:

1. (positive) rule context, contains variable declarations and application conditions;

[1]See section 2.1.4.4

2. match context for the positive part of the left rule side; stores variable assignments in compliance with the declarations of the rule context;
3. context for the *negative* part of the left rule side, contains attribute application conditions and variable assignments *additionally* to those made by the "positive match context". Additional variables can also be declared in the negative context. Variables declared *only* in the negative context cannot be employed in expressions on the right rule side;
4. match context for the negative part of the rule; "knows" all the variable values that were previously assigned in the positive match context and stores values for variables added within the negative rule context.

- transformation of attribute values according to previously generated rule-match mappings;
- applicability in a distributed environment, as described in section 2.3.3;

  For this, interface/local-relationships must be realized for:

  1. *Attribute tuple types*; an attribute tuple type that is the interface of other, local, types makes sure that all its attribute tuple declaration members are consistent (equal) with all of the corresponding local types' members. Members correspond when they have the same declaration name.
  2. *Attribute tuple instances*; an attribute tuple instance that is the interface of other, local, instances makes sure that all its members' values are equal to all of the corresponding local instance members.
  3. *Attribute Contexts*; the set of variables and their values is kept consistent for local contexts that share a common interface context.

  The client is responsible for connecting the right instances of, let's say, left hand side (LHS) of an interface rule to corresponding (parallel) instances of the LHS of a local rule. The type of the interface instance must be an interface type of the type of the local instance. The client is also responsible for correctly connecting the parallel contexts.

- persistence (save/restore services);

  It must be possible to store the state of all attribute entities and to restore them at a later stage.

- a non-blocking, event-driven graphical user interface for:

  – manipulating the attributes;

    An easy-to-use editor for creating attribute tuple types by adding and editing type members as well as entering expressions for the value tuple members must be provided.

- manipulating the variables, parameters and application conditions of rules;

  An editor for entering expressions and values for variables and Boolean expressions for application conditions should present the rule context state in a clear way.
- customizing of the attribute component;

  An editor for changing options of the attribute manager as well as those of every attribute handler must be available. The attribute manager, for example, can have a changeable list of available attribute handlers. Attribute handlers can have different standard editors for a given type. The Java expression handler has an extendable list of class packages to search in when looking for a class.

• facade pattern for simplifying user-specific use and integration into specific environments;

  In order to make the integration of the attribute component into client systems easier, facade interfaces[2] should be provided. The distinct frameworks to consider are:

  - attribute information access;

    This includes querying and setting information stored in attribute tuples and variables and parameters of rule contexts.
  - mapping and transformation of attributes;

    This is a facade for clients like Graph Transformation Systems or Petri Nets, that map attribute tuples and transform them accordingly, without being interested in the particular information stored therein.
  - calling and control of attribute editors;

    This is a facade for client Graphical User Interfaces that wish to easily integrate editors supplied by the attribute component.

The attribute component provides entities to be employed in a 1:1 relationship by *graph objects* [3]. These entities are called *attribute tuples*. Each *attribute tuple* has zero, one or more *attribute values* and is typed concerning the set of types and selectors of these values. Each *attribute value* has a type.

The attribute component is divided into two main parts. One of them, the *attribute manager*, is responsible for the *attribute tuples* and their types. The other, *attribute handler*, administers *attribute values* and *their* types. There can be more than one such *attribute handler*. Thus new types and functionality can by added in order to be employed for *attribute values*.

Frequently, several user views are required in a graphical system, when clarity of presentation is to be preserved in face of large quantities of information. More specific, the

[2]See page 39 for a description of the facade design pattern.

[3]"graph object" means either a node or an edge of a graph. The distinction is not essential in a general attributing concept like the one that is presented here.

hiding and moving of members within attribute tuples must be possible. So-called *view settings* are called for, where there is a view mask for every tuple type in the attribute component. View masks per tuple instance are not necessary.

The matching and transforming of attributes must not be affected by view manipulations. View settings are a service for implementers of editors and renderers, since permutation and hiding of entries in a list is a frequently required feature.

Three facade interfaces/classes with the following functionality are needed:

InformationFacade Creating and modifying attribute tuple types, setting/retrieving of tuple member values;

TransformationFacade Creating of rule and match contexts, adding/deleting of mappings in these contexts, performing transformations by applying match contexts;

EditorFacade Convenience methods for calling editors for various purposes from one class.

Chapter 5

System Design

This part presents the overall system structure. The interaction with external systems and the human user is shown. For the sake of clarity, each class diagram displays only the functionality that is relevant to the system aspect that it illustrates. The full functionality of each class can be found in appendix A.

5.1 Package overview

In order to keep the system manageable, the classes and interfaces are distributed in numerous packages, according to their role. The following overview lists all existing packages and briefly describes their contents.

5.1.1 Attribute Manager Packages

agg.attribute Interfaces of the attribute manager part, mostly containing database (attribute tuple) and graph transformation (rule context) functionality;

agg.attribute.impl Classes implementing the above interfaces;

agg.attribute.view Interfaces of the attribute manager part, containing view setting functionality;

agg.attribute.view.impl Classes implementing the above interfaces;

agg.attribute.gui Interfaces of the attribute manager part containing the GUI specification;

agg.attribute.gui.impl Classes implementing the above interfaces;

agg.attribute.gui.lang Isolating the language-dependant system messages;

agg.attribute.facade Facade interfaces, user-oriented collections of functionality to simplify the use of the attribute component.

agg.attribute.facade.impl Classes implementing the above interfaces.

5.1.2 Attribute Handler Packages

agg.attribute.handler Interfaces containing the database and graph transformation functionality of attribute handlers;

agg.attribute.handler.gui Interfaces containing the GUI specification for custom handler editors;

agg.attribute.handler.gui.impl Classes supporting the implementation of custom handler editors;

agg.attribute.handler.impl.javaExpr Implementation classes of the Java expression handler;

agg.attribute.handler.impl.javaExpr.gui Implementation classes of the Java expression handler's custom editors.

agg.attribute.parser.javaExpr Interpreter for Java expressions, used by the Java expression handler and containing a number of generated parser classes.

5.1.3 Other Packages

agg.attribute.test Classes implementing the attribute component's test environment;

agg.attribute.util Classes with supporting functionality.

5.2 Attributed Graphs

The attribute component provides entities to be employed in a 1:1 relationship by *graph objects* These entities are called *attribute tuples*. Each *attribute tuple* has zero, one or more *attribute values* and is typed concerning the set of types and selectors of these values. Each *attribute value* has a type.

The attribute component is divided into two main parts. One of them, the *attribute manager*, is responsible for and provides services for the *attribute tuples* and their types. The other, *attribute handler*, does the same for *attribute values* and *their* types. There can be more than one such *attribute handler*. Therefore, new types and functionality can by added in order to be used as *attribute values*.

5.2.1 Attribute Tuple and Attribute Members

The figure 5.1 shows how typing is achieved for attribute tuples, in addition to demonstrating the relationship between the attribute manager and the attribute handler. As extensions of *AttrTuple*, both *AttrType* and *AttrInstance* are aggregations of *AttrMember*. The interface requires the following assertions, which are hinted at by the class diagram layout:

- All members of *AttrType* are instances of *AttrTypeMember*.
- All members of *AttrInstance* are instances of *AttrInstanceMember*.

What follows is a brief description of the classes and methods in figure 5.1. More information about each class can be found in the appendix.

AttrTuple is the abstract root interface for all attribute tuples (AttrType, AttrInstance, AttrVariableTuple, AttrConditionTuple). A tuple consists of one or several members. It provides some services for access to tuple members. It also provides a common observable pattern to attribute objects, but this aspect will be discussed in section 5.6.

- public AttrMember getMemberAt(int index) Returns the member at the specified index;
- public AttrMember getMemberAt(String name) Returns the member with the specified name;
- public int getNumberOfEntries() Returns the number of members in the tuple.

AttrMember is the abstract root interface for all attribute members (AttrTypeMember, AttrInstanceMember, AttrVariableMember, AttrConditionMember).

- public AttrTuple getHoldingTuple() returns the tuple that contains this member;
- public String getName() returns the member name;
- public int getIndexInTuple() returns the member index within the containing tuple.

AttrType is the interface for an attribute tuple type. It extends the functionality of *AttrTuple* by methods for adding and deleting of members.

- public AttrTypeMember addMember() adds an empty declaration. The new declaration member is returned and can be extended by calling the respective AttrTypeMember methods;

- public void deleteMemberAt(String name) deletes the declaration (member) for the specified member name.
- public void deleteMemberAt(int index) deletes the declaration (member) at the specified index.

AttrTypeMember is the interface for a member of an attribute type.

- public void delete() removes itself from the tuple;
- public String getName() returns its name;
- public void setName(String name) sets the member name;
- public HandlerType getType() returns its type;
- public String getTypeName() returns the type name as string;
- public void setType(String typeName) sets the declaration type;
- public AttrHandler getHandler() returns the attribute handler;
- public void setHandler(AttrHandler h) sets the attribute handler.

AttrInstance is the interface for an attribute tuple instance.

- public AttrType getType() returns the type of the tuple instance;
- public void copy(AttrInstance source) copies the contents of an attribute instance into another.

AttrInstanceMember is the interface for an instance tuple member.

- public AttrTypeMember getDeclaration() returns the member type;
- public boolean isSet() tests if the value is set or not;
- public HandlerExpr getExpr() returns the expression (value) contained in this member;
- public void setExpr(HandlerExpr expr) sets the expression (value) of the member;
- public Object getExprAsObject() returns the member value as Object;
- public String getExprAsText() returns the textual representation of the expression.;
- public void setExprAsObject(Object value) sets the value of an instance member directly;
- public void setExprAsEvaluatedText(String expr) evaluates an expression and sets its value as this member's entry;
- public void setExprAsText(String expr) sets an expression for this member without immediate evaluation.

AttrManager is an abstract factory interface for creating attribute objects.

- public AttrHandler getHandler(String name) returns the attribute handler specified by name;
- public AttrHandler[] getHandlers() returns all registered attribute handlers;
- public AttrType newType() returns a new attribute tuple type;
- public AttrInstance newInstance(AttrType type) returns a new attribute tuple instance of the specified type.

HandlerType is the interface for types of attribute handlers.

- public String toString() returns the string representation of the type;
- public Class getClazz() returns the type as Java class.

HandlerExpr is the interface for expressions or values of attribute handlers.

- public String toString() returns the string representation of the expression;
- public HandlerExpr getCopy() returns a copy of the expression.

AttrHandler is the interface for the abstract factory of an attribute handler.

- public String getName() returns the name of the attribute handler;
- public HandlerType newHandlerType(String typeString) returns the handler type specified as string;
- public HandlerExpr newHandlerExpr(HandlerType type, String expr) returns a new handler expression of the specified type, the expression being specified as string;
- public HandlerExpr newHandlerExpr(HandlerType type, Object value) returns a new handler expression of the specified type, the expression being specified directly as a Java object.

The figure 5.2 displays the relationship between the graph objects and their attributes. The left side shows a typical graph transformation system, while the big high box on the right side contains parts of the attribute component. A graph transformation system typically has one *AttrManager* instance. A graph grammar has a collection of types for nodes and edges (GraphObjectType). Each of these types has exactly one instance of *AttrType*. Each graph in a graph grammar has zero, one or more graph objects (nodes, edges). Each of these objects has exactly one instance of *AttrInstance*.

The figure 5.3 shows a sample attribute tuple, taken from a screen shot of a tuple editor. The editor is explained in chapter 8. This figure just serves the purpose of visualizing how the concepts work in practice.

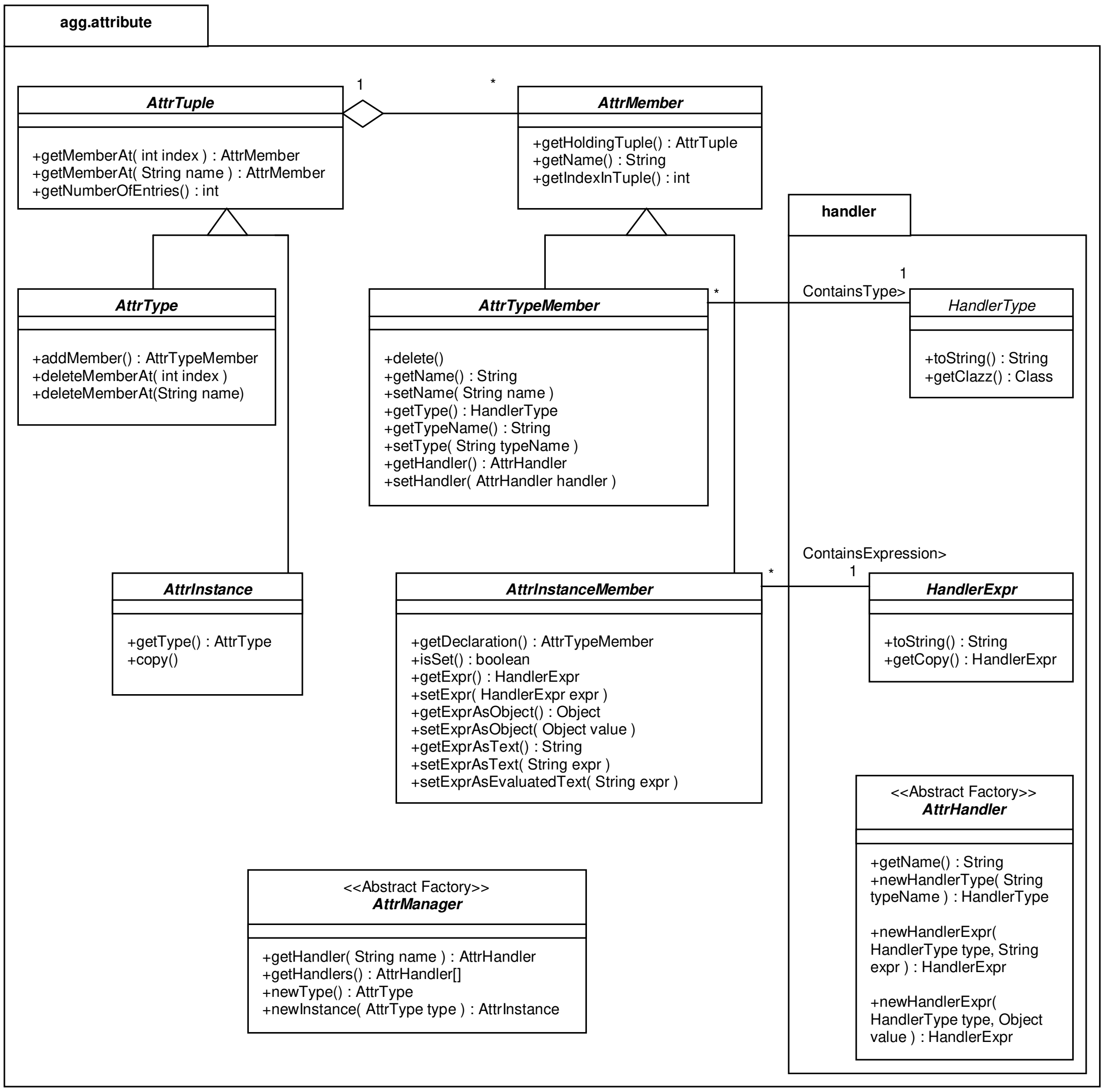

Figure 5.1: Attribute tuple and attribute members

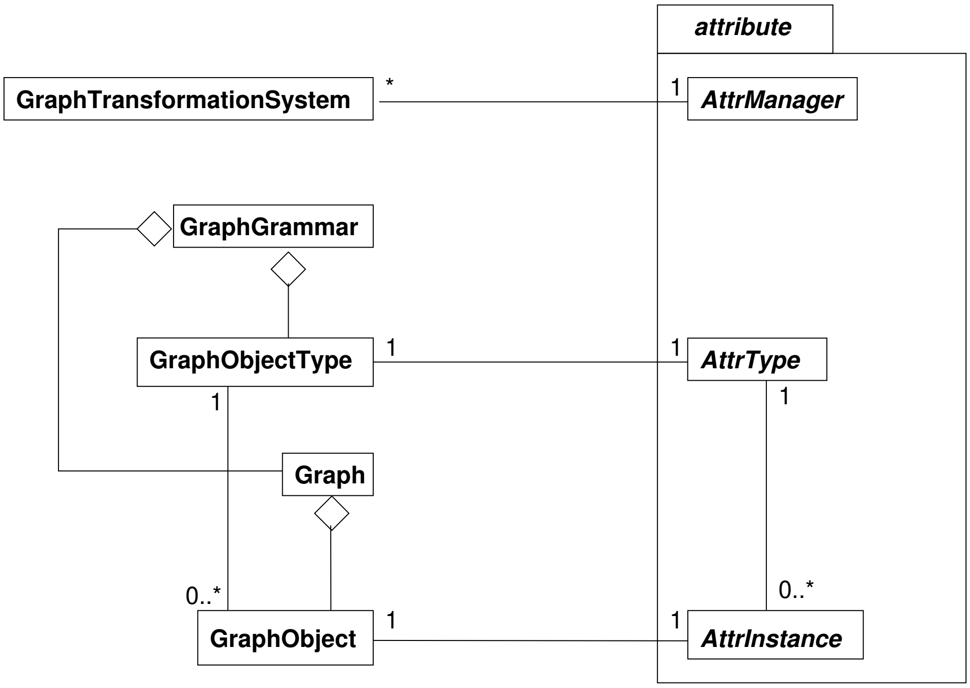

Figure 5.2: Relationship between the graph objects and their attributes

Handler	Type	Name	Expression
Java Expr	String	model	"VW Jetta"
Java Expr	double	length	4.35
Java Expr	String	id	"B-AE 753"
Java Expr	int	mileage	102070

Figure 5.3: The editor for attribute tuples

5.2.2 Attribute Values - Attribute Handler

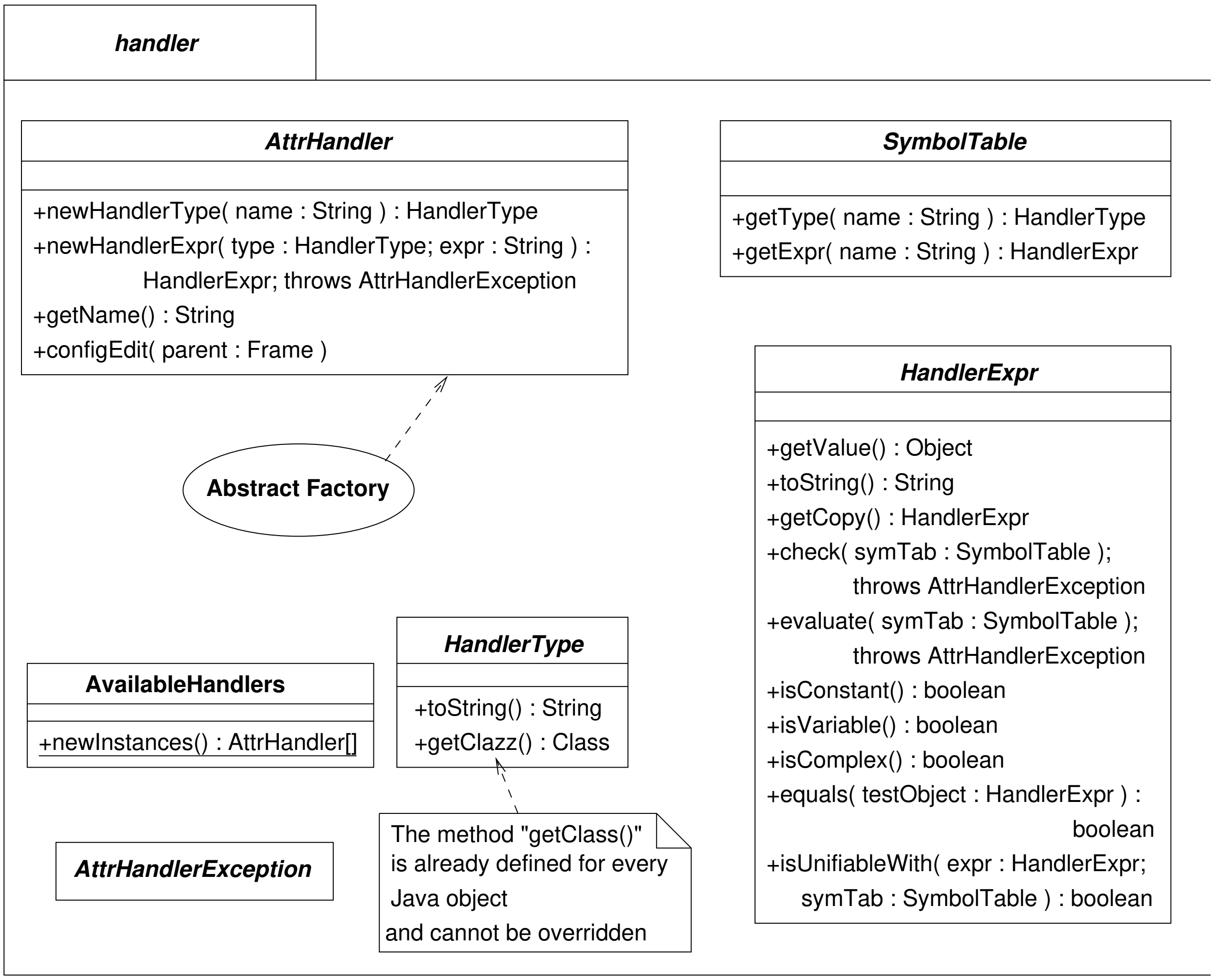

Figure 5.4: All interfaces of the handler package

The figure 5.4 shows all interfaces of the handler package. Similar to the attribute manager, the attribute handler also contains an *Abstract Factory*, the interface *AttrHandler*.

Short description of the interfaces and methods of this package:

AttrHandler is the interface for the abstract factory of an attribute handler.

- public String getName() returns the name of the attribute handler;
- public HandlerType newHandlerType(String typeString) returns the handler type specified as string;
- public HandlerExpr newHandlerExpr(HandlerType type, String expr) returns a new handler expression of the specified type, the expression being specified as string;

- public HandlerExpr newHandlerExpr(HandlerType type, Object value) returns a new handler expression of the specified type, the expression being specified directly as a Java object.
- public void configEdit(Frame parent) is an obsolete method for calling a customizing editor. For the GUI concepts see section 5.9; see section 6.3 about how to invoke the customizing editors and section 8.2.3 for a manual of the provided customizing editors.

HandlerType is the interface for types of attribute handlers.

- public String toString() returns the string representation of the type;
- public Class getClazz() returns the type as Java class.

HandlerExpr is the interface for expressions or values of attribute handlers.

- public String toString() returns the string representation of the expression;
- public Object getValue() returns the value as a Java object;
- public HandlerExpr getCopy() returns a copy of the expression;
- public void check(SymbolTable symTab) throws AttrHandlerException type-checks the expression under a given symbol table with declarations;
- public void evaluate(SymbolTable symTab) evaluates the expression under a given symbol table containing variable declarations and values;
- public boolean isConstant() checks if the expression is constant;
- public boolean isVariable() checks if the expression is a single variable;
- public boolean isComplex() checks if the expression is a complex one (like x+1);
- public boolean equals(HandlerExpr testObject) returns 'true' if the expression equals 'testObject', 'false' otherwise;
- public boolean isUnifiableWith(HandlerExpr expr, SymbolTable symTab) checks if the recipient can be "matched", i.e. "unified" with the first parameter under a certain variable assignment.

SymbolTable is an interface for type checking and expression evaluating. It serves the communication between the Attribute Manager and the Attribute Handlers. It contains types and values of identifiers. This interface is implemented by the Attribute Manager. A Handler evaluates expressions; the values are assigned by the manager, never by the handler; therefore, there are no assigning methods in this interface. It contains the following two methods:

- public HandlerType getType(String name) returns the type of the specified identifier;

- public HandlerExpr getExpr(String name) returns the value of the specified identifier.

AvailableHandlers The purpose of this class, being the only class in a package of interfaces, is that an attribute manager knows where to find its handlers. Whenever a new handler is installed, its fully qualified pathname has to be added to the static array 'nameList'. That's all a new handler has to do besides, of course, implementing those methods. All an attribute manager has to do is (besides implementing the SymbolTable interface) calling 'newInstances()'. It then gets an array of attribute handler instances, one for every handler in the mentioned list.

This class has one static variable and one static method:

- protected static String nameList[] = { "agg.attribute.handler.impl.javaExpr.JexHandler" } This list can be extended to include further attribute handlers. Afterwards the AvailableHandlers class has to be recompiled.
- static public AttrHandler[] newInstances() returns an array of attribute handler instances, one for every handler in nameList. This method is typically called by the attribute manager at the beginning of a session (startup).

The interfaces in this package have to be implemented by any new attribute handler implementation, with two exceptions:

- *SymbolTable* is implemented by the attribute manager and is used as lookup-table for variable values when evaluating expressions[1];
- AvailableHandlers is not an interface but a class which allows the registering of new handler implementations.

[1]In the concrete realization, this interface is implemented by the class **attribute.impl.ContextView**.

5.3 Support of Different Views

Figure 5.5 shows how the interface aspects dealing with selective views. The attribute tuples offer the possibility of selecting their members by index. Moreover, if an instance of *AttrViewSetting* is supplied, the returned members will be selected according to that specific view (mask). A view setting can be seen as a collection of masks, where there is exactly one mask for each attribute tuple type. The interface *AttrViewSetting* offers methods for hiding/showing and moving the members within a tuple for each attribute type. If *null* is supplied as view setting when addressing members of a tuple, the default view setting is taken.

AttrViewSetting is designed as a mediator interface, facilitating view-dependent access to attribute objects. The "Mediator" design pattern (cf. [GHJV95], 273ff) was chosen for a loose and lightweight coupling of attribute objects and their visual representation. It also allows view-dependent (editor) and view-independent (graph transformation unit, database) users of the attribute component to identify their attribute objects by the same instance handles. Please note that the integer selectors for attribute tuple members are not absolute indexes as in the *AttrTuple* interface. Rather, they are "slots", member positions with respect to this view.

There can be an arbitrary number of views, each holding exactly one (changeable) representation of an attribute tuple (order of members, hiding of members).

Each view has two *subviews*, an *open view* and a *masked view*. They basically share the same tuple layout information, with one exception. Tuple access using the sub-view obtained by calling getOpenView() does not hide the 'hidden' members, although they can be hidden by invoking setVisibleAt(aTuple, false, aSlot). The hiding effect only occurs when using the sub-view obtained by calling getMaskedView().

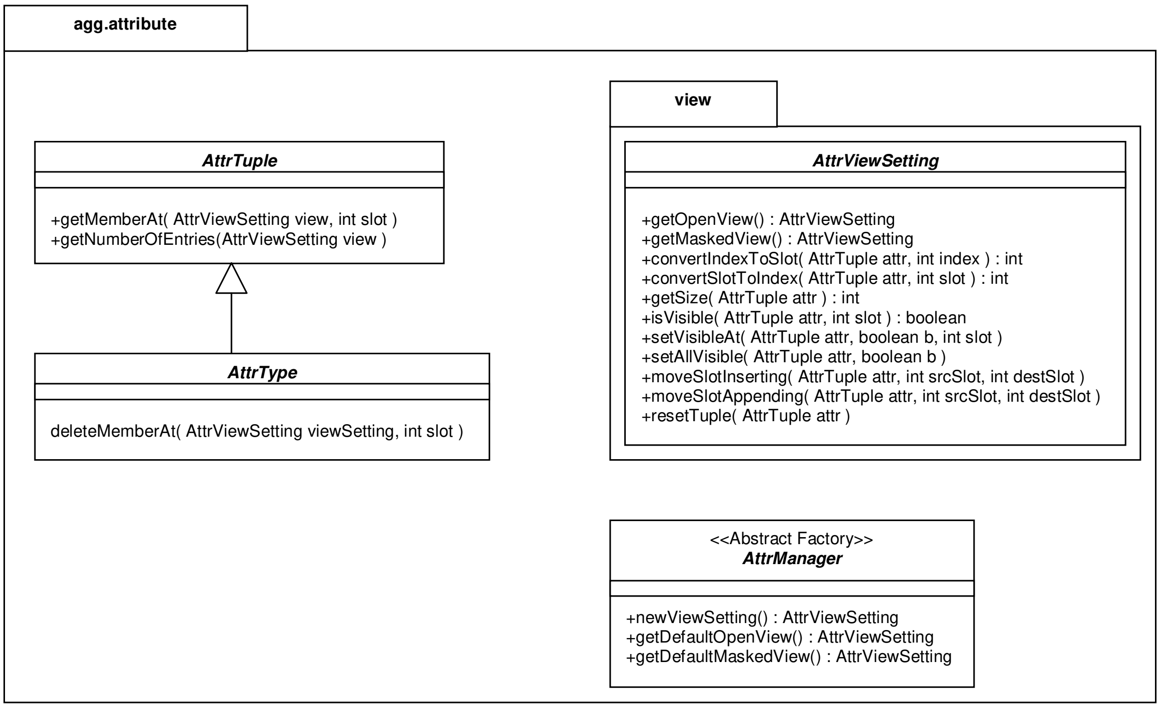

Figure 5.5: View support for attribute editors and renderers

5.4 Rule and Match Context

Graph rules have sets of attribute variables and application conditions (equations). To realize the aspect that all attribute expressions within a rule share the same set of these variables and conditions, the entity *AttrContext* is introduced. All attribute tuples within the same rule have access to the same context. Figure 5.6 displays the information structure contained within an *AttrContext*.

5.4.1 Context services

A brief description of classes that are participating in the context framework displayed in figure 5.6:

AttrContext embodies the framework for allocation of variables, administration of attribute mappings and application conditions in rules.

- public AttrConditionTuple getConditions() Returns the application conditions tuple of this context;
- public AttrVariableTuple getVariables() Returns the variable tuple of this context;
- public boolean doesAllowComplexExpressions() Checks if complex expressions are permitted in this context;
- public boolean doesAllowNewVariables() Checks if implicit variable declarations are permitted in this context;
- public boolean doesAllowEmptyValues() Checks if empty attribute values are permitted in this context;
- public void setAllowVarDeclarations(boolean isAllowed) Sets if implicit variable declarations are permitted in this context;
- public void setAllowComplexExpr(boolean isAllowed) Sets if complex expressions are permitted in this context;
- public void setAllowEmptyValues(boolean isAllowed) Sets if empty attribute values are permitted in this context;
- public void freeze() Switching on the freeze mode; mapping removals are deferred until 'defreeze()' is called; some match search algorithms require this feature;
- public void defreeze() Performs mapping removals which were delayed during the freeze mode;
- public HandlerType getType(String name) Getting the type of an identifier;
- public HandlerExpr getExpr(String name) Getting the value of an identifier.

The methods doesAllowComplexExpressions, doesAllowNewVariables, doesAllowEmptyValues(), setAllowComplexExpr, setAllowVarDeclarations and setAllowEmptyValues allow very specific configuration of what expressions are permitted in a context. For a graph transformation system, there should be no need to use these services, since there are factory methods in *AttrManager* that set the appropriate access rights for the left and right rule sides, respectively.

The methods getType and getExpr fulfill the *agg.attribute.handler.SymbolTable* interface. Thus, an instance of *AttrContext* can be supplied as a symbol table to the type-checking and evaluation methods in agg.attribute.handler.HandlerExpr, see section 5.2.2.

AttrInstance is the interface for an attribute instance tuple.

- public AttrContext getContext() returns the context of this instance tuple;
- public int getNumberOfFreeVariables(AttrContext context) returns the number of variables declared by this instance, which have no value assigned to them yet. Each variable name is counted only once, even if it is used more than once in this tuple.

AttrVariableTuple is the interface for a variable tuple. Its services are described in section 5.4.3.

AttrConditionTuple is the interface for a condition tuple. Its services are described in section 5.4.4.

AttrMapping represents the mapping between two attribute instances. It also keeps record of variables that were assigned values because of the mapping.

- public boolean next() in case when there are more than one possibilities for an attribute match between two instance tuples (like in the case of regular expression matches), this method applies the next possible match.
- public void remove() dissolves the mapping; removes variable assignments made by this mapping from its context and dissolves the connection between the attribute instances.

AttrManager is the abstract factory interface. In the framework of attribute contexts, it provides methods for creating of rule and match contexts.

- public AttrContext newContext(int mapStyle) creates a new attribute context which is the root of a context tree;
- public AttrContext newContext(int mapStyle, AttrContext parent) creates a new attribute context which extends an existing one. In this way, a match context is created with its rule context as 'parent'.

- public AttrContext newLeftContext(AttrContext context) creates a left rule side view for an existing rule context. Here, variables can be declared, but the assignment of complex expressions to single attribute values is forbidden.
- public AttrContext newRightContext(AttrContext source) creates a view for an existing rule context, through which variables cannot be declared; complex expressions as attribute values are allowed, but only declared variables may be used.
- public void checkIfReadyToMatch(AttrContext ruleContext) throws AttrException checks if matching can be performed with respect to a given rule context. If the rule context in question is without inconsistencies, this method remains 'silent'. Otherwise, it throws an exception whose message text describes the reason.
- public AttrMapping newMapping(AttrContext mappingContext, AttrInstance source, AttrInstance target) throws AttrMatchException creates a mapping between two attribute instances; the mapping is performed according to the context mapping property (match/plain) and is integrated into the context;
- public void checkIfReadyToTransform(AttrContext matchContext) throws AttrException Checking if a transformation can be performed with the attributes with respect to a given context. If the match context in question is complete and without inconsistencies, this method remains 'silent'. Otherwise, it throws an exception whose message text describes the reason.

The 'mapStyle' for the context creating methods has the following possibilities:

AttrMapping.PLAIN_MAP In Graph Transformation: rule mapping;

AttrMapping.MATCH_MAP In Graph Transformation: match mapping.

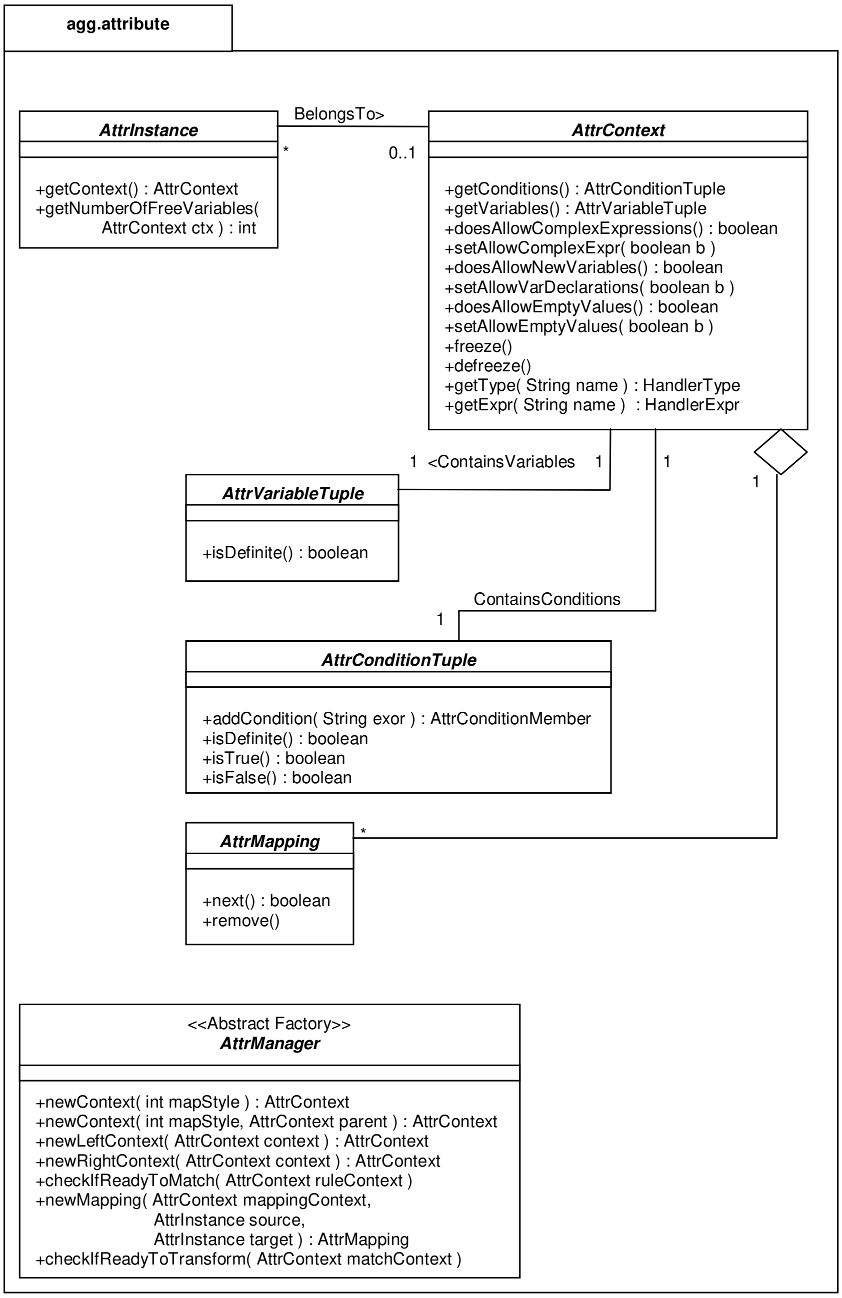

Figure 5.6: Context: variables, conditions and mappings

5.4.2 Relationship between contexts, rules and matches

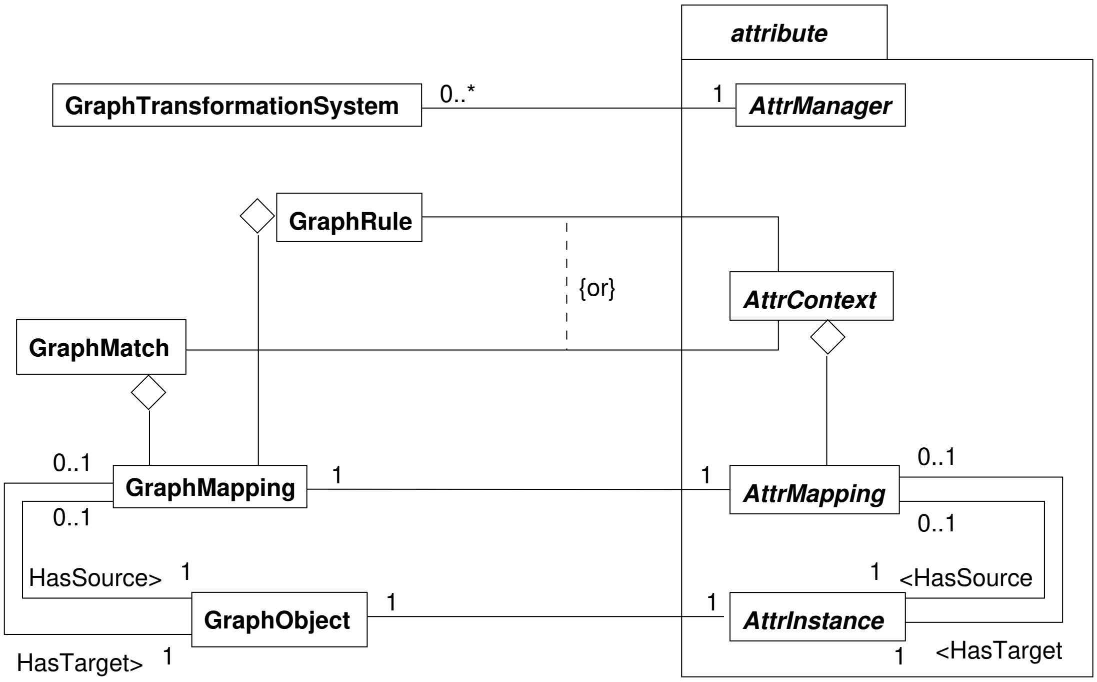

Figure 5.7: Using contexts in graph rules and matches

The figure 5.7 shows the relationship between the graph system and attribute contexts of rules and matches.

The figure 5.8 shows how each graph rule side has a separate relationship with the rule context. This allows for attributing the relation by access rights, so that different handling of attribute expressions (left side declares and references variables, the right side only references them; complex expressions are not allowed on the left side) can be realized.

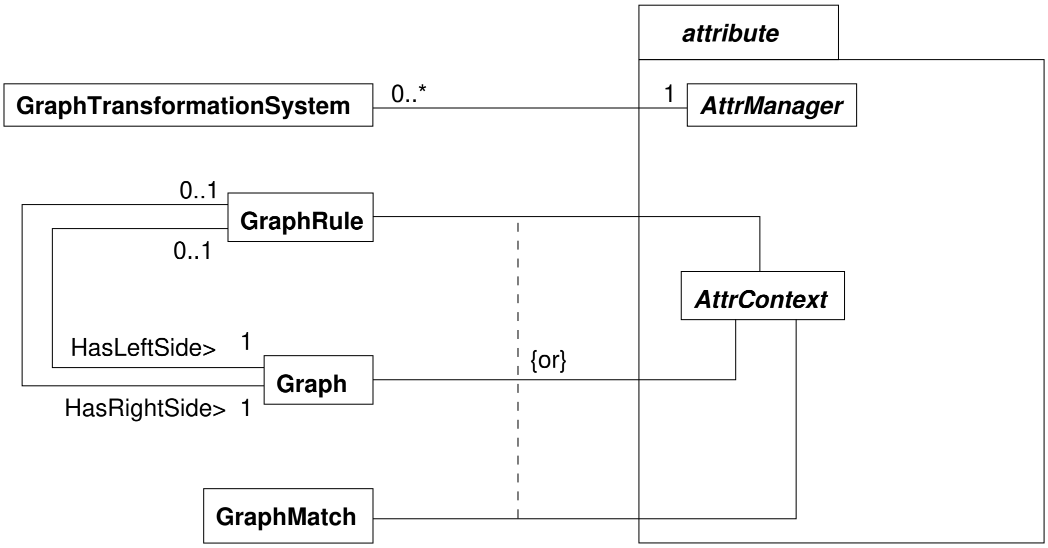

Figure 5.8: Separated context access for each rule side

5.4.3 Rule Parameters and Variables

The figure 5.9 shows the concept of variables in an attribute context. Parameters are realized as variables with certain properties (see also methods isInputParameter(), isOutputParameter(), setInputParameter(boolean), setOutputParameter(boolean) in the *AttrVariableMember* interface).

AttrVariableTuple is the interface for a variable tuple.

- public boolean isDefinite() tests if all variables evaluate to definite values.

Methods for adding and deleting of declaration members are accessible via the interface *AttrType*, whose instance can be obtained using the method getType(), inherited from the *AttrInstance* interface.

Methods for access to the variable tuple members are inherited from the *AttrTuple* interface.

AttrVariableMember is the interface for a variable tuple member.

- public void delete() Removes this member from its tuple.
- public boolean isDefinite() Tests if the variable evaluates to a definite value.
- public boolean isInputParameter() Tests if this variable is an IN-parameter.
- public void setInputParameter(boolean b) Sets, if the variable is to be an IN-parameter.
- public boolean isOutputParameter() Tests if this variable is an OUT-parameter.
- public void setOutputParameter(boolean b) Sets, if the variable is to be an OUT-parameter.

The methods for querying and setting other aspects of the variable declaration (handler, type, name) are accessible via the interface *AttrTypeMember* whose instance can be obtained using the method getDeclaration(), inherited from the *AttrInstanceMember* interface.

The methods for access to the variable expression or value are also inherited from the *AttrInstanceMember* interface.

The interfaces *AttrTuple*, *AttrMember*, *AttrType*, *AttrTypeMember*, *AttrInstance* and *AttrInstanceMember* are described in section 5.2.1.

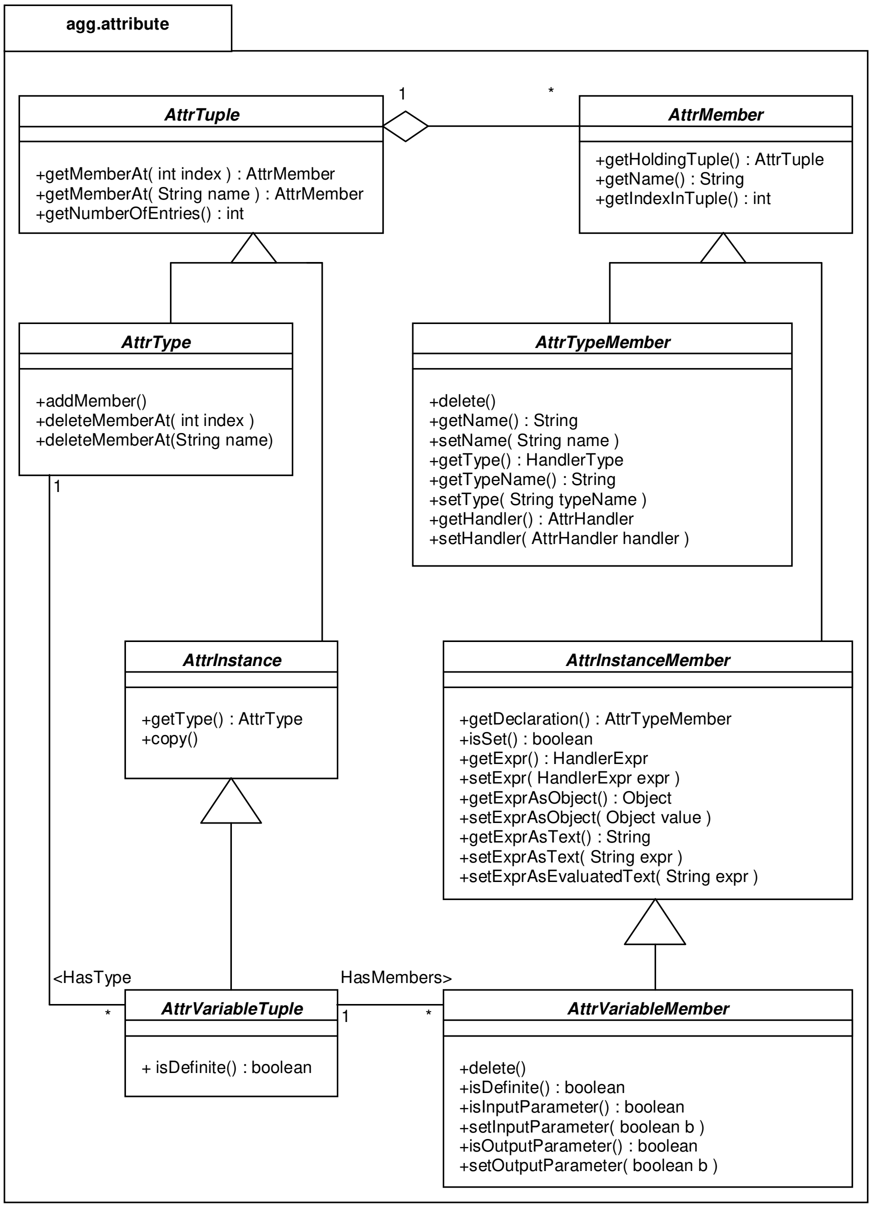

Figure 5.9: Variables of an attribute context

5.4.4 Application Conditions

The figure 5.10 shows the concept of attribute conditions. Attribute application conditions of a rule are realized as an attribute tuple (*AttrConditionTuple*), where all members are of type *AttrConditionMember*, meaning that they contain Java expressions of type *boolean*, like `x == 5` or `title != "Mr"`.

AttrConditionTuple is the interface for a condition tuple.

- public AttrConditionMember addCondition(String expr) creates a condition member and returns it. This is a convenience method that makes sure the type is 'boolean' and sets the expression 'expr' as text, without immediate evaluation. For deleting of a condition tuple member, use delete() in *AttrConditionMember*.
- public boolean isDefinite() Tests if all members can yield true or false.
- public boolean isTrue() Tests if AND-ing of all members yields true.
- public boolean isFalse() Tests if the tuple contains members which can be evaluated and yield 'false'.

Methods for access to the condition tuple members are inherited from the *AttrTuple* interface. Deleting of condition tuple members is performed using the delete() method in *AttrConditionMember*.

Please note that the methods isTrue() and isFalse() are not necessarily negations of each other.

AttrConditionMember is the interface for a condition tuple member.

- public void delete() Removes this member from its tuple.
- public boolean isDefinite() Tests if the expression can yield true or false.
- public boolean isTrue() Tests if the expression yields true.
- public boolean isFalse() Tests if the expression yields false.

Please note that the methods isTrue() and isFalse() are not necessarily negations of each other. When isDefinite() returns 'false', both isTrue() and isFalse() return false.

The methods for access to the conditions expressions are inherited from the *AttrInstanceMember* interface.

The interfaces *AttrTuple*, *AttrMember*, *AttrType*, *AttrTypeMember*, *AttrInstance* and *AttrInstanceMember* are described in section 5.2.1.

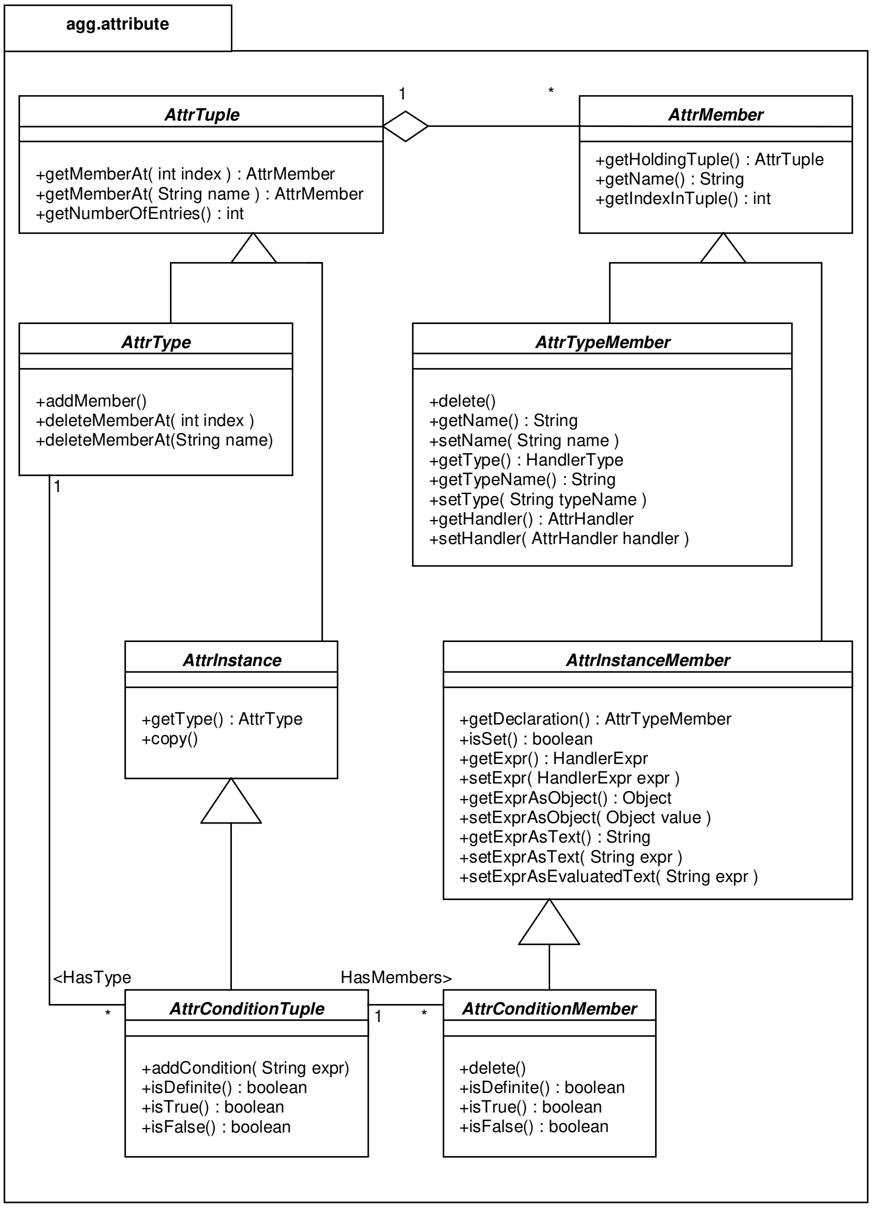

Figure 5.10: Application conditions in an attribute context

5.4.5 Attribute Mapping

The *AttrMapping* interface is used to manage mappings between attribute tuples. This occurs in:

- rule definitions, where left side objects are mapped onto right side objects;
- match constructions, where the left side objects of a rule are mapped onto objects in the working graph.

In the case of rule definitions, the mapping contains just the source and the target attribute tuple as information. A match mapping additionally retains the information about the variables that it binds to certain values. This is required by match search algorithms that can use back-jumping when a dead-end is reached because of an earlier variable binding.

The figure 5.11 shows the information structure of attribute mappings. Each attribute mapping contains one source and one target attribute tuple instance as well as a container with those variable names that are bound by this mapping.

AttrMapping represents the mapping between two attribute instances. It also keeps record of variables that were assigned values because of the mapping.

- public boolean next() in case when there are more than one possibilities for an attribute match between two instance tuples (like in the case of regular expression matches), this method applies the next possible match.
- public void remove() dissolves the mapping; removes variable assignments made by this mapping from its context and dissolves the connection between the attribute instances.

AttrManager is the abstract factory interface. In the framework of attribute mappings, it provides a method for creating of a new mapping.

- public AttrMapping newMapping(AttrContext mappingContext, AttrInstance source, AttrInstance target) throws AttrMatchException creates a mapping between two attribute instances; the mapping is performed according to the context mapping property (match/plain) and is integrated into the context;

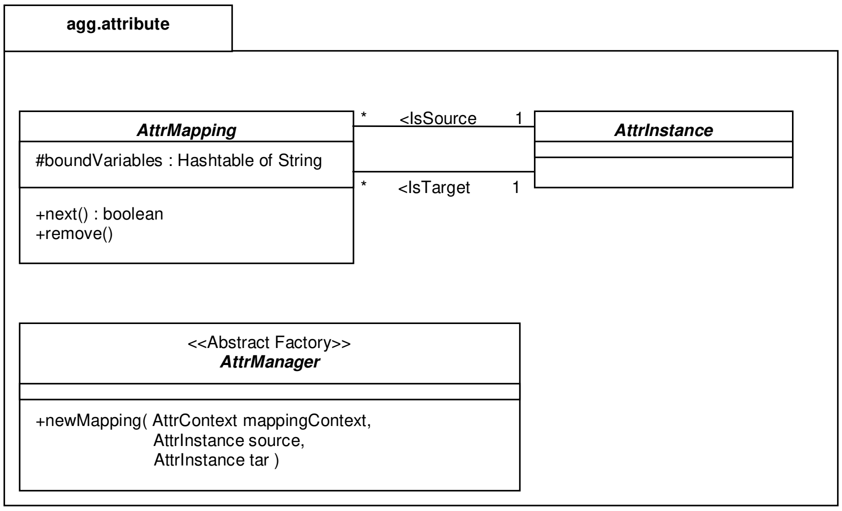

Figure 5.11: Attribute mappings in an attribute context

5.5 Attribute Transformation

AttrInstance is the interface for an attribute instance tuple.

- public void apply(AttrInstance rightSide, AttrContext context) applies a rule; the substitutions occur "in-place", i.e. in the recipient; In Graph Transformation, this method is applied to attributes of host graph objects, "rightSide" being an attribute of the right side of the rule and "context" being the "match"-context built up by subsequently matching the attributes of corresponding graphical objects.

  In connection with the Java expression handler, where arbitrary user-implemented classes can be involved, a transformation success cannot be guaranteed. In case where expression evaluation fails, the Java exception is propagated to the graph transformation unit (client) that invoked the attribute transformation. The exception is casted to RuntimeException when it is passed to the client. In order to have access to exception-specific functionality, the exception should be casted back to its original class.

AttrContext is the interface for an attribute context.

When a working graph is being transformed according to a rule whose left side was previously matched with the working graph, the graph transformation algorithm can proceed in two manners:

In-place-replacement The method apply(rightSideTuple, matchContext) is called for each of the target attribute tuples, where rightSideTuple is the right side tuple which is associated with the tuple to be transformed, and matchContext is the context of the match which contains all mappings from the left rule side into the working graph.

Diagram-replacement The graph transformation algorithm copies the objects that are to be transformed into a new graph and proceeds as above, calling the apply(...) method on the copies.

For an example on how to build contexts, create mappings and invoke transformation of attributes, see section 6.2, where the Transformation Facade Interface is employed.

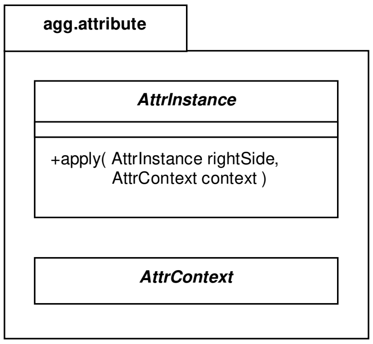

Figure 5.12: Classes and methods involved in attribute transformation

5.6 Event-Driven Communication and The Subject-Observer Pattern

The attribute component realizes an event-driven update mechanism. This is required by the following features:

Event-driven GUI The standard graphical user interfaces (GUI) for complex systems today are event-driven, providing a maximum of convenience and efficiency. In order to be able to integrate the GUI of the attribute component into another system's GUI, the best way is to make it event-driven. Some examples of what would trigger an event are given below:

- Editing of attribute expressions can affect the fulfillment of application conditions in the respective graph transformation rule. The change can immediately be made visible.
- Editing of an attribute tuple type (adding/removing of members, changing the type of a member etc.) affects the representation of all tuple instances.
- When visualizing graph transformation (in a debugging/tracing-like mode), attribute changes can be displayed immediately.

Distribution The interdependence between interface attribute tuples and their local instances should be realized using the subject-observer pattern.

The subject-observer pattern is realized on two levels:

- Attribute tuple level, propagating just events concerned with intrinsic information;
- View level, propagating events connected to visualization changes, which include all the events of the attribute tuple level.

The figure 5.13 shows how the subject-observer pattern is designed for attribute tuples. Clients not interested in visualization (order of members, hiding) register as *AttrObserver* and implement two methods:

- attributeChanged(AttrEvent event), which defines the observer's reaction to an event from the attribute entity that it is observing.
- isPersistentFor(AttrTuple at), which decides if the link to the observer is preserved when saving/restoring the system state. Typically, clients using the attribute component as a data base should return true.

The figure 5.14 shows how the subject-observer pattern is designed for attribute tuple views. Clients interested in visualization (order of members, hiding), typically editors, register as *AttrViewObserver* and implement the method attributeChanged(AttrViewEvent event).

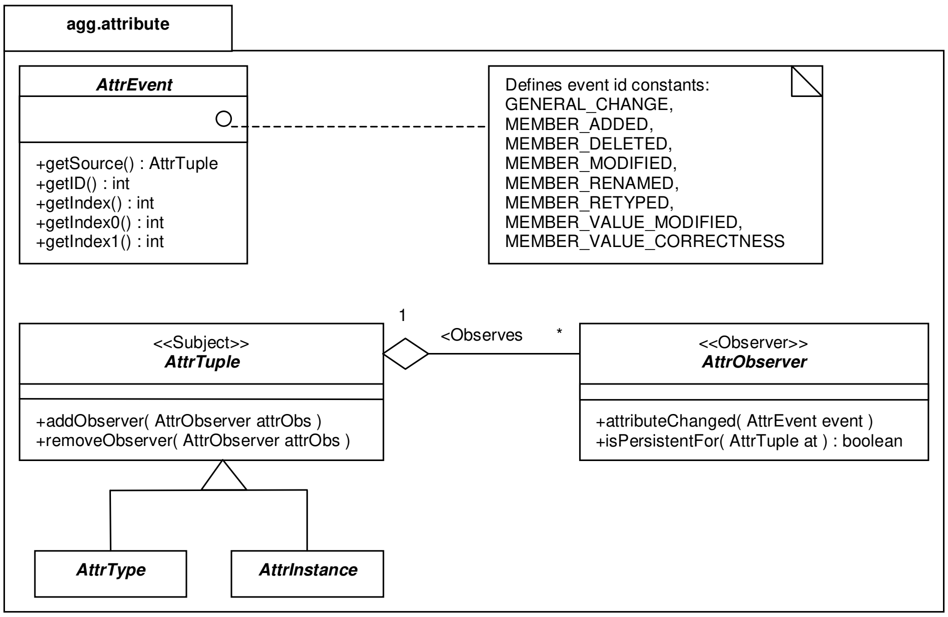

Figure 5.13: Realization of the subject-observer pattern for attribute tuples

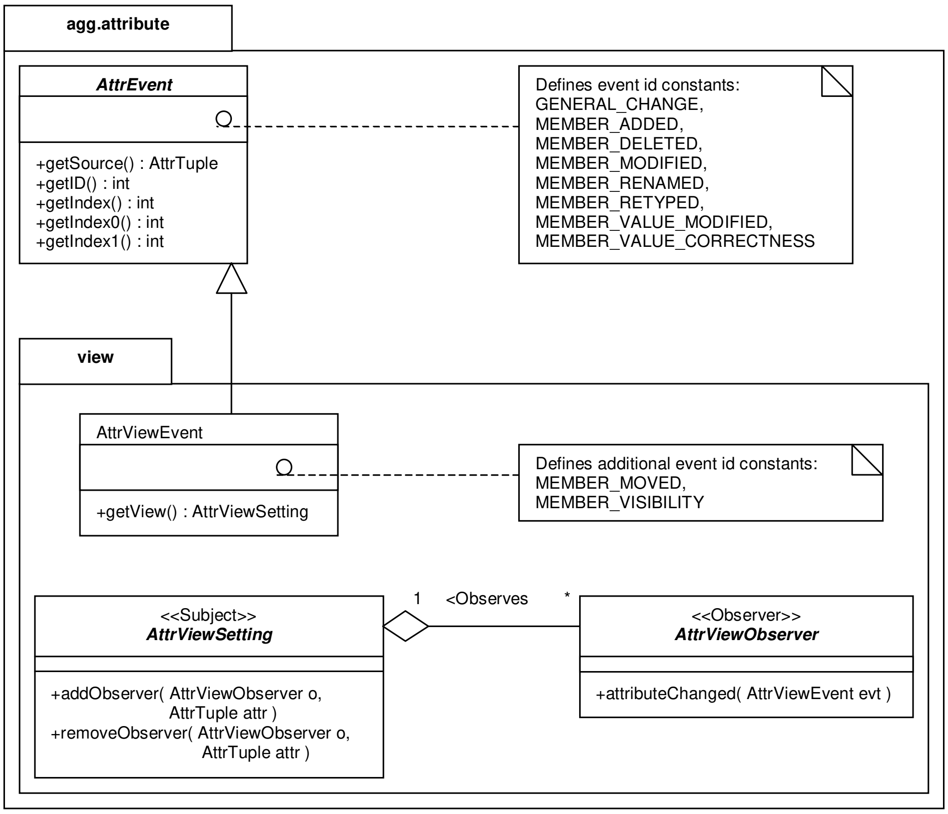

Figure 5.14: Realization of the subject-observer pattern for attribute tuple views

5.7 Distribution: Interface vs. Local Tuples and Contexts

The connection between interface tuples and local tuples is realized using the subject-observer mechanism described in the previous section.

The interfaces participating in the distribution framework are shown in figure 5.15.

AttrDistributionBroker is a Mediator interface for distribution purposes. It provides services for creating and maintaining of interface/local-relations between attribute tuples and contexts.

The "connect" methods establish relationships between attribute entities that ensure the consistency between interface entities and their local entities.

- public void connect(AttrType interfaceType, AttrType localType) Makes a type tuple into an interface of another type tuple. The ensured consistency conditions for an interface and all of its local entities are:
  1. A local type tuple is valid when it contains a valid corresponding declaration member for all of its interfaces' members. A corresponding member is one that has the same name as in the local type tuple. The member is valid when it also has the same handler and type.
  2. Adding a new declaration member to the interface type tuple: If the declaration does not exist in a local type tuple, it is also added there.
  3. Deleting a declaration member from the interface type tuple: The corresponding declaration members stay in the local parts.
  4. Changing a declaration member's type in the interface type tuple: The corresponding local members' types are changed accordingly.
  5. Changing a declaration member's name in the interface type tuple: The corresponding local members' declarations remain unchanged.
- public void disconnect(AttrType interfaceType, AttrType localType) Ends a type tuple's role as an interface of another type tuple.
- public void connect(AttrInstance interfaceInstance, AttrInstance localInstance) Makes an instance tuple into an interface of another interface tuple. The ensured consistency conditions for an interface and all of its local entities are:
  1. When an interface instance member value is changed, the corresponding local members get the same value too.
  2. When a local instance member value that has a corresponding member in an interface tuple is changed, that interface member and all of its related local members get the same value too.

- public void disconnect(AttrInstance interfaceInstance, AttrInstance localInstance)
  Ends an instance tuple's role as an interface of another interface tuple.
- public void connect(AttrContext interfaceContext, AttrContext localContext)
  Makes a context into an interface of another context. The ensured consistency conditions for an interface and all of its local entities are:
  1. The types and values of the context variables fulfill the consistency conditions for type tuples and instance tuples, respectively.
  2. The condition expressions fulfill the consistency conditions of instance tuples.
- public void disconnect(AttrContext interfaceContext, AttrContext localContext)
  Ends a context's role as an interface of another context.

An example that illustrates how to use attribute distribution in distributed graph transformation is shown in section 6.4.

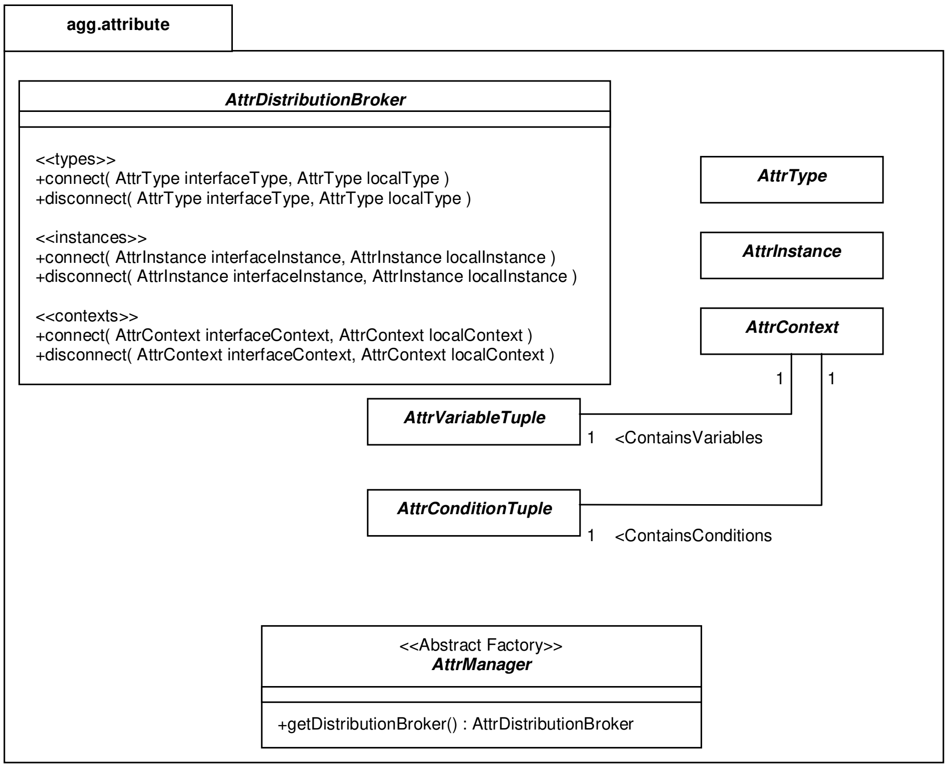

Figure 5.15: Distribution framework

AttrManager is the abstract factory interface. In the framework of distribution, it provides a method for obtaining the attribute distribution broker.

- public AttrDistributionBroker getDistributionBroker() Returns the distribution broker.

AttrContext embodies the framework for allocation of variables, administration of attribute mappings and application conditions in rules. It is described in section 5.4.1

AttrType is the interface for an attribute type tuple.

AttrInstance is the interface for an attribute instance tuple.

AttrVariableTuple is the interface for a variable tuple. Its services are described in section 5.4.3.

AttrConditionTuple is the interface for a condition tuple. Its services are described in section 5.4.4.

5.8 Attribute Handler

The extendibility of the attribute component gains considerably from the possibility of including additional attribute handlers. Each attribute handler can offer one to many types. Each member of an attribute tuple is typed by one of such a type. Expressions (values) to these types can be created and used in attribute tuple instance members. The common interface makes sure that information needed by the graph transforming unit (e.g. if the expression is a variable, a constant or a complex expression) can be queried, and the expressions can be transformed using a symbol table (variable binding).

5.8.1 Exchanging Information and Values

Different attribute handlers can use each other's services and integrate them into their own functionality. For once, they can create instances of another handler's expressions and manipulate them using the common *HandlerExpr* interface's methods. Moreover, and this is very interesting for graph transformation systems, expressions of a one attribute handler can include expressions from another. This is done using variables. Let's say there is a handler *ColorHandler* providing the type *Color*. Then, if a parameter $c : Color$ is given, a tuple member typed *int* (using the Java expression handler) can refer to c, for example querying it's blue value (according to RGB), using a method implemented by *Color*: c.getBlue().

5.8.2 Java Expression Handler

The attribute component contains the Java Expression Handler that allows access to all Java types and classes. It makes the system additionally flexible and extendable. Instead of having to implement a complete attribute handler for each new type, the type can be implemented as a Java class and subsequently used in expressions within the existing Java Expression Handler. The accepted expressions correspond to the Java grammar, with the following exceptions:

- No operators with side effects (`=, +=, ++` etc.);
- Additional feature: chaining of method calls, returning the addressed instance, *not* the method call return value; invocation using the `;` operator instead of the `.` operator:

  ```
  aVector;addElement(elem1);addElement(elem2);addElement(elem3)
  ```

  appends the instances `elem1, elem2, elem3` to `aVector` and returns the modified `aVector`. This feature should be used with care, since it produces side-effects and in graph transformation, the order of transformed graph objects matters. To avoid undetermined grammars, copying (`aVector`, in the above example) before application can be used.
- The implemented path searching algorithm expects the class paths to be enclosed in `$`. Therefore, the Java class

  ```
  java.lang.System.out
  ```

  has to be expressed as

  ```
  $java.lang.System$.out
  ```
- Differences in precision of Primitive Java types are not implemented. Although all the primitive types are accepted by the parser, internally all integer numbers are mapped to `int` and all floating numbers to `float`.

Everything else is exactly as in Java expressions. Even conditional expressions are handled, e.g.:

```
a >= max ? aVector.elementAt( a ) : null
```

The EBNF-like notation that is used below follows the input syntax of the parser generator tool *JavaCC*:

"LITERAL" Literals are enclosed by double quotes;

(Expr) Grouping in parentheses

[**OptionalExpression**] Zero or one occurence;

(ZeroOrMoreOccurences)* Zero, one or more occurences;

Expr1 | Expr2 | .. | Expr$_n$ Alternatives;

Expr1 Expr2 .. Expr$_n$ Chain;

The terminal symbols follow the JavaCC syntax and are self-explained. The implemented Java handler accepts and processes expressions that are described by the following grammar:

```
CompilationUnit          ::= Expression <EOF>
Expression               ::= ConditionalExpression
ConditionalExpression    ::= ConditionalOrExpression
                               ["?" Expression ":" ConditionalExpression]
ConditionalOrExpression ::= ConditionalAndExpression
                               ( "||" ConditionalAndExpression )*
ConditionalAndExpression ::= InclusiveOrExpression
                               ( "&&" InclusiveOrExpression )*
InclusiveOrExpression    ::= ExclusiveOrExpression
                               ( "|" ExclusiveOrExpression )*
ExclusiveOrExpression    ::= AndExpression ( "^" AndExpression )*
AndExpression            ::= EqualityExpression ( "&" EqualityExpression )*
EqualityExpression       ::= RelationalExpression
                               (   "==" RelationalExpression
                                 | "!=" RelationalExpression )*
RelationalExpression     ::= AdditiveExpression
                               (   "<" AdditiveExpression
                                 | ">" AdditiveExpression
                                 | "<=" AdditiveExpression
                                 | ">=" AdditiveExpression )*
AdditiveExpression       ::= MultiplicativeExpression
                               (   "+" MultiplicativeExpression
                                 | "-" MultiplicativeExpression )*
MultiplicativeExpression ::= UnaryExpression
                               (   "*" UnaryExpression
                                 | "/" UnaryExpression
                                 | "%" UnaryExpression )*
UnaryExpression          ::=  "-" UnaryExpression
```

```
                          | "+" UnaryExpression
                          | UnsignedUnaryExpression
UnsignedUnaryExpression ::=  "~" UnaryExpression()
                          | "!" UnaryExpression
                          | PrimaryExpression
PrimaryExpression       ::=  "new" FullClassName() "["
                          | PrimaryPrefix() ( PrimarySuffix() )*
PrimaryPrefix           ::=  Literal
                          | Id
                          | "$" ClassName "$"
                          | "(" Expression ")"
                          | AllocationExpression
PrimarySuffix           ::=  ArrayIndex()
                          | Member()
                          | Action()
ArrayIndex              ::=  "[" Expression() "]"
Action                  ::=  ";" MemberName() ArgumentList()
Member                  ::=  ( MemberName "(" [ ArgumentList ] ")" )
                          | MemberName
ArgumentList            ::= Expression ( "," Expression )*
DimensionList           ::= ( "[" "]"
                          | "[" Expression() "]" )+
AllocationExpression    ::= "new" FullClassName() ArgumentList()
ArrayAllocation         ::= "new" FullClassName() DimensionList()
MemberName              ::= <IDENTIFIER>
FullClassName           ::= ClassName() | "$" ClassName() "$"
ClassName               ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
Id                      ::= <IDENTIFIER>
Literal                 ::=   <INTEGER_LITERAL>
                           | <FLOATING_POINT_LITERAL>
                           | <CHARACTER_LITERAL>
                           | <STRING_LITERAL>
                           | <FLOATING_POINT_LITERAL>
                           | BooleanLiteral
                           | NullLiteral
BooleanLiteral          ::= "true" | "false"
NullLiteral             ::= "null"

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
```

```
      | <OCTAL_LITERAL> (["l","L"])? >
| < DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
| < HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
| < OCTAL_LITERAL: "0" (["0"-"7"])* >
| < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"] >
| < EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
| < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
| < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
| < #LETTER: [ "a"-"z", "A"-"Z", "_" ] >
| < #DIGIT: [ "0"-"9"] >
```

5.9 Graphical User Interface

The GUI of the attribute component was designed in such a way that it can easily be integrated into other systems' GUIs, like graph transformation editors, database editors, Petri-Net-editors etc. This is done using the JavaBeans standard interfaces. They allow e.g. to query the size and implemented features of a GUI component (sub-editor).

5.9.1 Attribute Tuples

A hierarchy of editor interfaces exists in the package agg.attribute.gui for attribute tuples, rule and match contexts and options. This is shown in figure 5.16.

AttrEditor is an abstract interface, parent of all attribute editor interfaces; it provides basic operations like setting and getting of the attribute and the editor manager.

- public Component getComponent() returns the AWT component (window entity) that can be integrated in a larger GUI context or displayed separately;
- public AttrManager getAttrManager() returns the attribute manager.
- public void setAttrManager(AttrManager m) sets the attribute manager.
- public AttrEditorManager getEditorManager() returns the editor manager.
- public void setEditorManager(AttrEditorManager m) sets the editor manager.

AttrTupleEditor is the interface for an editor of an attribute tuple. Some instances thereof can be restricted to attribute tuple instances (AttrInstance), others can have manipulative access to the tuple type as well.

- public void setTuple(AttrTuple anAttrTuple) sets the attribute tuple to be edited.
- public AttrTuple getTuple() returns the edited attribute tuple.
- public void setViewSetting(AttrViewSetting anAttrViewSetting) sets the view for the editor.
- public AttrViewSetting getViewSetting() returns the editor's view.

AttrContextEditor is and editor interface for an attribute context, allows editing of variables and application conditions of a (rule/match) context.

- public void setContext(AttrContext anAttrContext) sets the edited context.
- public AttrContext getContext() returns the edited context.

AttrCustomizingEditor is the interface for a customizing editor of an attribute manager as well as of his handlers. This interface has no methods.

AttrTopEditor is the most comprehensive editor interface for the attribute component. Its (inherited [2]) services allow to manipulate an attribute context, an attribute tuple (instance and its type) and to customize the options of the attribute manager and handlers.

[2] Although on the interface level, it looks like multiple inheritance, the implementation uses delegation.

AttrEditorManager Abstract factory interface for attribute editors.

- public AttrTopEditor getTopEditor(AttrManager m, AttrViewSetting v) creates a comprehensive editor which allows to manipulate an attribute context, an attribute tuple (instance and its type) and to customize the options of the attribute manager and handlers.
- public AttrTupleEditor getSmallEditorForInstance(AttrManager m, AttrViewSetting v, AttrInstance inst) creates a new small editor for an attribute tuple instance, where only the values can be changed. The editor window component can be integrated into graphical objects which hold the attributes.
- public HandlerEditorManager getHandlerEditorManager() returns the manager (abstract factory) for handler editors.

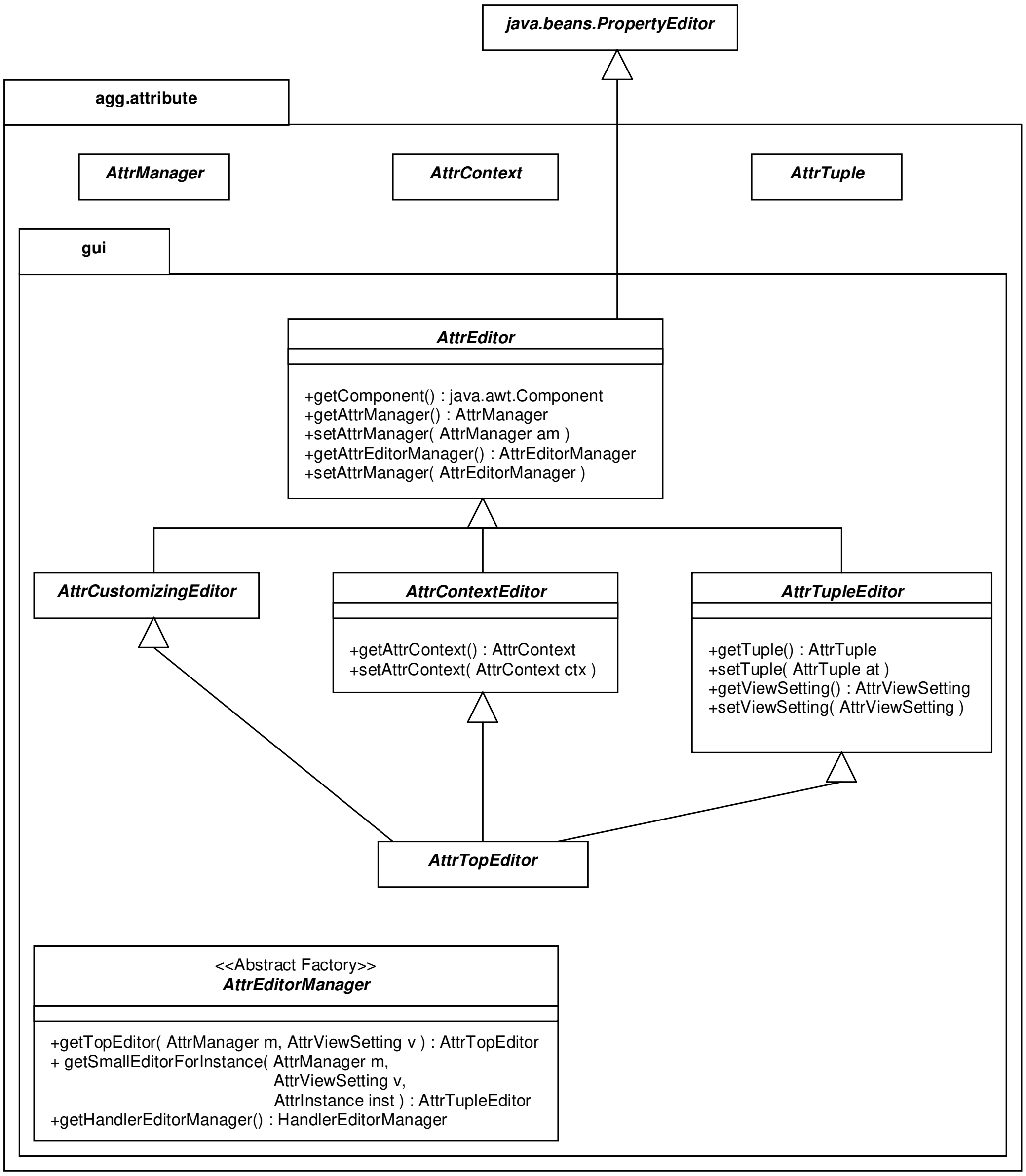

Figure 5.16: Editor hierarchy for the attribute manager

5.9.2 Attribute Values - Custom Editors

Custom editors can be included into the GUI on a per-handler and a per-type basis. The interfaces to be implemented for such cases are shown in figure 5.17.

HandlerEditor This interface is extended by handler editors, so clients can be notified of occurring changes.

- public void addEditorObserver(HandlerEditorObserver obs) adds a new editor observer to the editor.
- public void removeEditorObserver(HandlerEditorObserver obs) removes an editor observer from the editor.

HandlerEditorObserver This interface must be implemented be clients who wish to listen to changes in a handler editor.

- public void editingStopped(HandlerChangeEvent e) This method is invoked when the GUI user stopped editing.
- public void editingCancelled(HandlerChangeEvent e) This method is invoked when the GUI user cancelled editing.

HandlerTypeEditor This interface is implemented by attribute handler type editors; it provides services for the attribute type (tuple) editors.

- public Component getRendererComponent(AttrHandler handler, HandlerType typeToRender, Dimension availableSpace) returns a graphical component for displaying the specified type. The 'availableSpace' limit should be honoured, since this is a service for displaying the type in a table cell. However, the renderer can contain tools (e.g. buttons) for invoking its larger custom renderer. Either 'handler or 'typeToRender' cannot be null.
- public Component getEditorComponent(AttrHandler handler, HandlerType typeToEdit, Dimension availableSpace) returns a graphical component for editing the specified type. The 'availableSpace' is a recommendation when the editor wishes to be operable in a compact table cell and needs not be taken into account. Either 'handler or 'typeToRender' cannot be null.
- public HandlerType getEditedType() returns the edited type.

HandlerExprEditor This interface is implemented by attribute handler expression editors; it provides services for the Attribute instance (tuple) editors.

- public Component getRendererComponent(HandlerType type, HandlerExpr exprToRender, Dimension availableSpace) Returns a graphical component for displaying the specified expr. The 'availableSpace' limit should be honored, since this is a service for displaying the expr in a table cell. However, the renderer can contain tools (e.g. buttons) for invoking its larger custom renderer. Either 'type' or 'exprToRender' cannot be null.
- public Component getEditorComponent(HandlerType type, HandlerExpr exprToEdit, Dimension availableSpace) Returns a graphical component for editing the specified expr. The 'availableSpace' is a recommendation when the editor wishes to be operable in a compact table cell and needs not be taken into account. Either 'type' or 'exprToEdit' cannot be null.
- public HandlerExpr getEditedExpr() Returns the edited expression.

HandlerCustomizingEditor This interface allows to interactively customize an attribute handler.

- public Component getComponent() Returns a graphical component for customizing the handler (e.g. setting and changing options).
- public AttrHandler getAttrHandler() Returns the edited attribute handler.
- public void setAttrHandler(AttrHandler handler) Sets the edited attribute handler.

HandlerEditorManager

- public HandlerCustomizingEditor getCustomizingEditor(AttrHandler handler) Returns the customizing editor for an attribute handler.
- public HandlerTypeEditor getTypeEditor(AttrHandler handler, HandlerType type) Returns the standard type editor for a certain handler and start editing the specified type.
- public HandlerExprEditor getExprEditor(AttrHandler handler, HandlerType type, HandlerExpr expr) Returns the instance editor for a handler type and start editing the specified expression.

HandlerChangeEvent Implementations of this interface signal changes in a handler editor.

- public HandlerEditor getSourceEditor() Returns the editor that signaled the change.

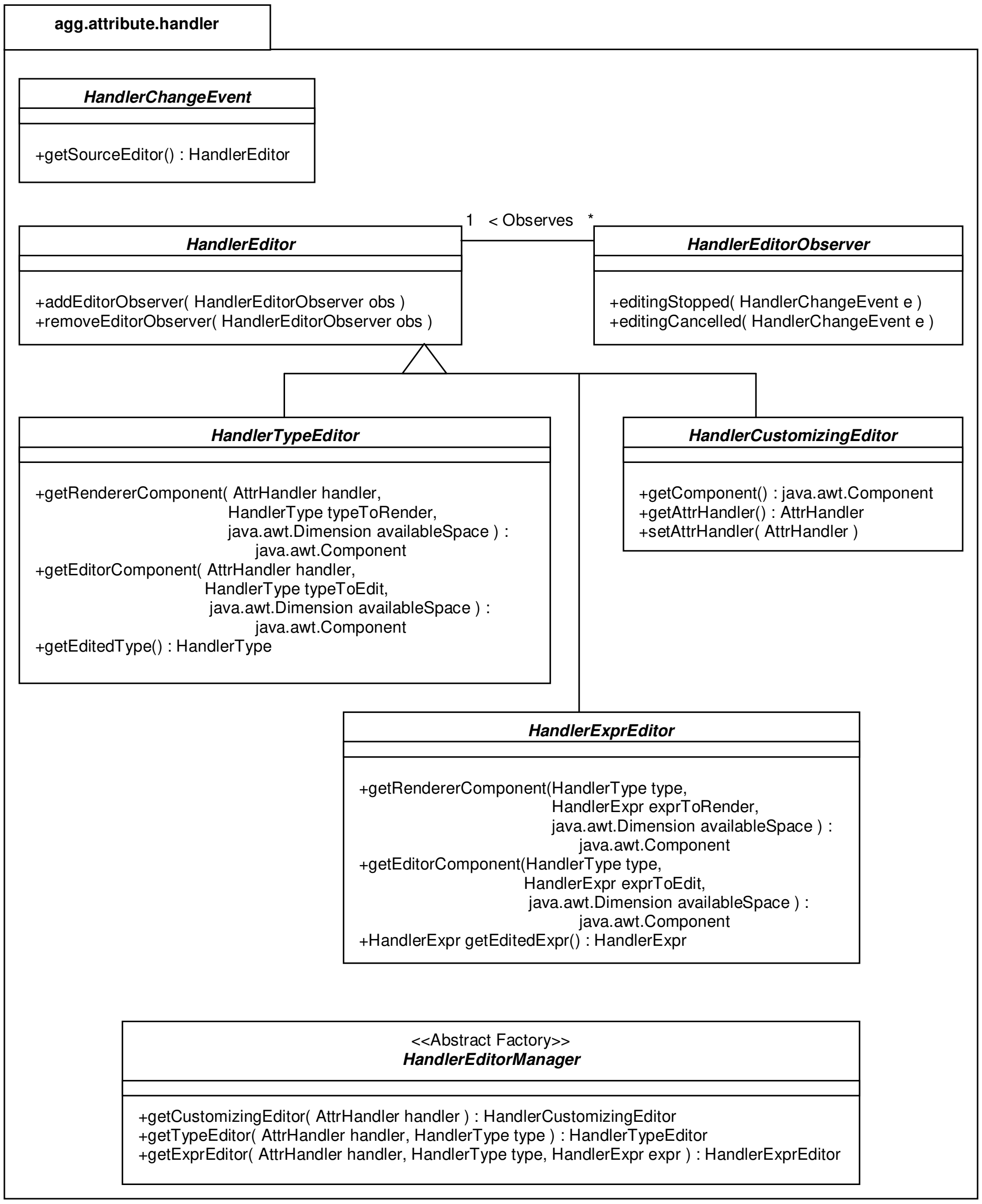

Figure 5.17: Interfaces for custom handler editors

Chapter 6

Integrating and Employing in Client Applications

This chapter describes how client systems can use structures and services offered by the attribute component. Three major frameworks are supported by the present system, and a facade interface is provided for each of them in the agg.attribute.facade.InformationFacade package, which is shown in figure 6.1 The frameworks are:

Information framework includes those system parts that manage the information contained in the attribute tuples and their members; the operations typically perform the creating of tuples, setting and retrieving of member values etc.;

Transformation framework provides infrastructure for graph transformation which uses graph grammar rules; it can also be used for other sorts of transformation, e.g. for Petri Nets;

Editor framework supplies various editors which can be integrated into a client system's graphical user interface.

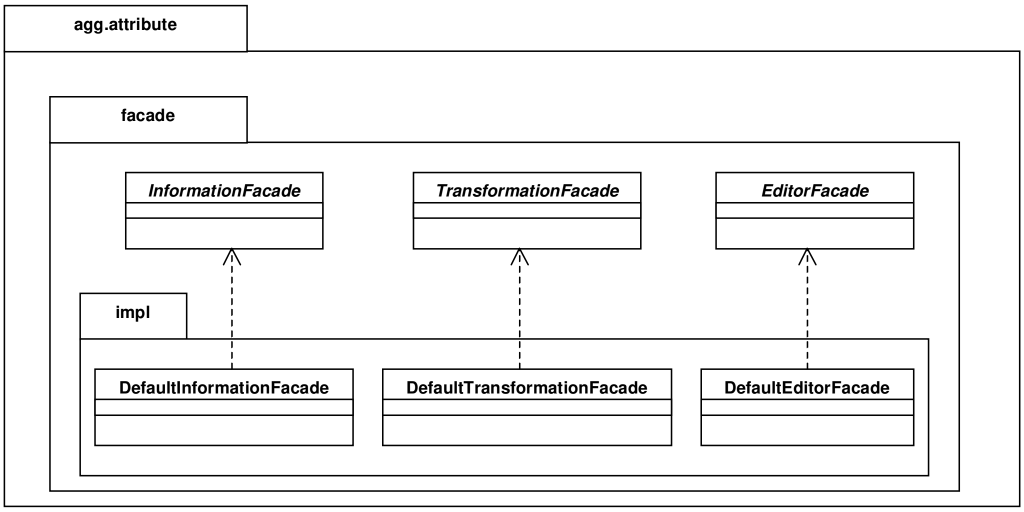

Figure 6.1: Framework Facades

The facade interfaces are geared for implementers of attribute component clients, who are not yet familiar with the system. The facades make general use of the system easier and clearer, and provide a convenient point of departure for exploring more specific system features. The latter is done by looking at the facade implementations in the package agg.attribute.facade.impl and learning where certain tasks are offered. For example, if one wished to change the name of a tuple member, he would look in the *InformationFacade* interface and find the method:

```
/** Setting a member type name. */
public void setName( AttrTypeMember memberDecl,
                     String memberName );
```

Now, by inspecting the implementation in DefaultInformationFacade:

```
/** Setting a member type name. */
public void setName( AttrTypeMember memberDecl,
                     String memberName ){
    memberDecl.setName( memberName );
}
```

one can see that the task is delegated to `memberDecl`, which is an instance of a class implementing the *AttrTypeMember* interface. This means that similar (and possibly more specific) operations are offered in the *AttrTypeMember* interface.

The three frameworks are described in detail in the following sections.

6.1 Information Framework

The interface *InformationFacade* in the package agg.attribute.facade is a convenience interface, being a collection of methods for storing and retrieving information in attribute tuples. It's implementation would typically delegate the tasks by calling methods of interfaces *AttrManager*, *AttrTuple*, *AttrMember*, *AttrType*, *AttrTypeMember*, *AttrInstance* and *AttrInstanceMember*. Figure 6.2 displays the methods of the *InformationFacade* interface. It also shows the interfaces implicitly involved.

The detailed description of the methods can be found in the appendix. What follows is an example of how to use the interface methods in order to create a tuple type, an instance of that type and to fill the instance with some data.

```
///////////////////////////////////////////////////////
// Example of using the 'InformationFacade' interface:
```

```
//////////////////////////////////////////////////////

import agg.attribute.*;
import agg.attribute.facade.*;
import agg.attribute.facade.impl.DefaultInformationFacade;
import agg.attribute.handler.*;

// Getting the handle for the information facade:
InformationFacade infoFacade = DefaultInformationFacade.self();

// Creating a new attribute tuple type:
AttrType tupleType = infoFacade.createTupleType();

// Registering an observer (event listener) of the type:
// infoFacade.addObserver( tupleType, typeNode )

// Getting the handle for the Java handler:
AttrHandler javaHandler = infoFacade.getJavaHandler();

// Creating member declarations for 'type', such that it
// describes a conference:
infoFacade.addMember( tupleType, javaHandler, "String", "location" );
infoFacade.addMember( tupleType, javaHandler, "String", "date" );
infoFacade.addMember( tupleType, javaHandler,
      "int", "no of participants");
infoFacade.addMember( tupleType, javaHandler, "double", "budget" );

// Creating a new instance of 'type' without a context:
AttrInstance tupleInstance =
infoFacade.createTupleInstance( tupleType, null );

// Registering an observer (event listener) of the instance:
// infoFacade.addObserver( tupleInstance, instNode );

// Setting values of the tuple members:
AttrInstanceMember instMember;
instMember = infoFacade.getInstanceMemberAt( tupleInstance, "location" );
infoFacade.setExprAsObject( instMember, "Berlin" );
instMember = infoFacade.getInstanceMemberAt( tupleInstance, "date" );
infoFacade.setExprAsObject( instMember, "Nov 1st, 1999" );
instMember = infoFacade.getInstanceMemberAt( tupleInstance,
       "no of participants" );
infoFacade.setExprAsEvaluatedText( instMember, "2000" );
```

```
instMember = infoFacade.getInstanceMemberAt( tupleInstance, "budget" );
infoFacade.setExprAsEvaluatedText( instMember, "50000.00" );

// End of example
//////////////////////////////////////////////////////////
```

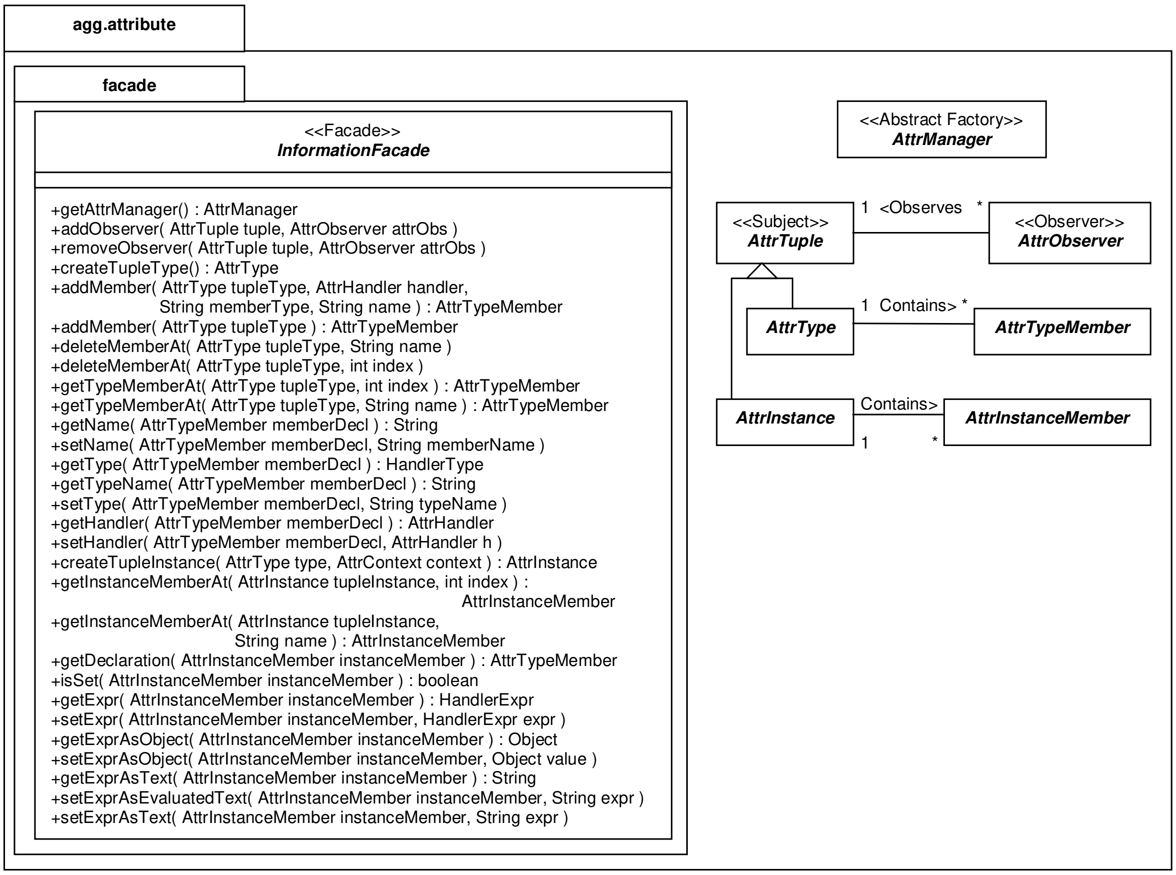

Figure 6.2: Information Facade

6.2 Transformation Framework

The interface *TransformationFacade* in the package agg.attribute.facade is a convenience interface for clients interested in rule-based attribute transformation. It's implementation would typically delegate the tasks by calling methods of interfaces *AttrManager*, *AttrContext* and *AttrMapping*. Figure 6.3 displays the methods of the *TransformationFacade* interface. It also shows the interfaces implicitly involved.

The detailed description of this facade's methods can be found in the appendix. Here an example of using this facade interface:

```
/////////////////////////////////////////////////////
// Example of using the 'TransformationFacade' interface:
/////////////////////////////////////////////////////

import agg.attribute.*;
import agg.attribute.facade.*;
import agg.attribute.facade.impl.DefaultTransformationFacade;

// Getting the handle for the transformation facade:
TransformationFacade transFacade = DefaultTransformationFacade.self();

// Creating a new rule context:
AttrContext ruleContext = transFacade.newRuleContext();

// Creating contexts for both rule sides:
AttrContext leftContext = transFacade.getLeftContext( ruleContext );
AttrContext rightContext = transFacade.getRightContext( ruleContext );

// Creating contexts for the negative application conditions (NACs):
AttrContext negativeRuleContext0 =
transFacade.getNegativeRuleContext( ruleContext );
// AttrContext negativeRuleContext1 = ...

// ... creating attribute tuple instances on both rule sides,
//     manipulating their values, editing variables and application
//     conditions of the rule context ...

// Mapping left side attribute tuple instances to those
// on the right side, within the rule context:
AttrMapping ruleMapping =
    transFacade.newMapping( ruleContext, sourceTuple, targetTuple );
```

```
// Mapping left side attribute tuple instances to those
// in the negative application condition:
AttrMapping ruleNegativeMapping =
    transFacade.newMapping( negativeRuleContext0,
                            sourceTuple, targetTuple );

// When an attribute mapping is to be dissolved:
transFacade.removeMapping( ruleMapping );

// Checking, if the rule context is consistent, i.e. there are no
// contradictory equations in the application conditions:

try{
    transFacade.checkIfReadyToMatch( ruleContext );
}
catch( AttrException ex ){
    System.out.println( ex.getMessage());
    return;
}

// Creating a match context:
AttrContext matchContext = transFacade.newMatchContext( ruleContext );

// Mapping left side tuple instances to those in the work graph,
// within the match context. Hereby, an attempted mapping can fail,
// whereupon the cause can be analyzed by the match searching algorithm:

AttrMapping matchMapping;
try{
    matchMapping =
        transFacade.newMapping( matchContext, sourceTuple, targetTuple );
}
catch( AttrMatchException ex ){
    int id = ex.getID();
    if( id == AttrMatchException.VARIABLE_BINDING ){
        AttrInstance firstBindingTuple = ex.getFirstBindingTuple();
        // ...
    }
}

// Creating a negative match context:
AttrContext negMatchContext =
```

```
transFacade.newNegativeMatchContext( negativeRuleContext0,
     matchContext );

// When an attribute handler allows more than one match, e.g. when
// matching regular expressions, all possibilities may be successively
// tried:
transFacade.nextMapping( matchMapping );

// When an attribute mapping is to be dissolved:
transFacade.removeMapping( matchMapping );

// A match searching algorithm might require information to use in
// its heuristics; a good estimation value is the number of free variables
// of an attribute tuple instance:
int numOfFreeVariables =
    transFacade.getNumberOfFreeVariables( sourceTuple, matchContext );

// Checking, if the match context is consistent and transformation
// of attributes can be performed:

try{
    transFacade.checkIfReadyToTransform( matchContext );
}
catch( AttrException ex ){
    System.out.println( ex.getMessage());
    return;
}

// Performing attribute transformation:
transFacade.apply( workGraphTuple, rightRuleSideTuple, matchContext );

// End of example
//////////////////////////////////////////////////////////
```

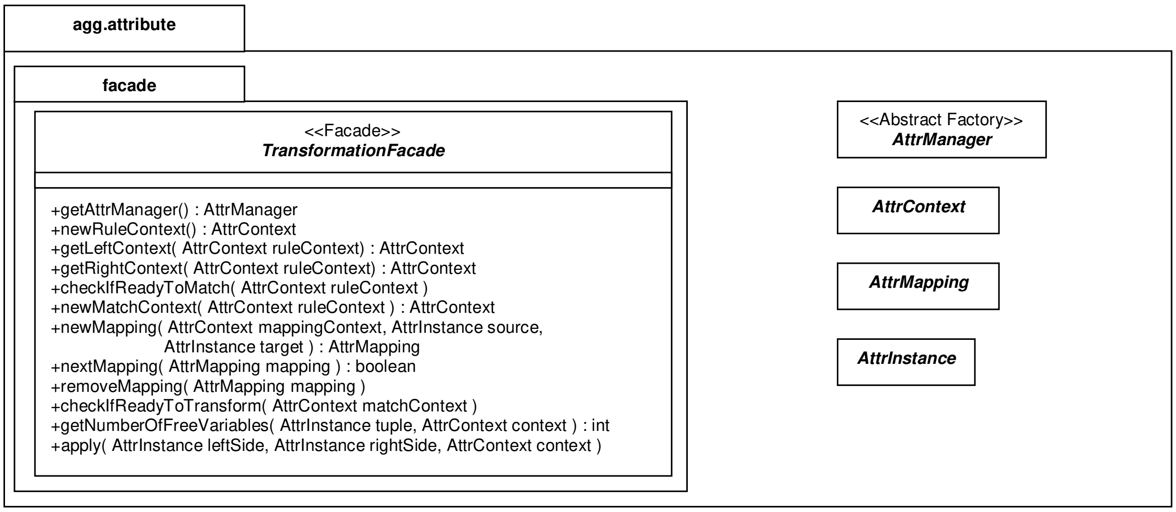

Figure 6.3: Transformation Facade

6.3 Editor Framework

The interface *EditorFacade* in the package agg.attribute.facade is a convenience interface for integrating attribute component's editors into larger graphical user interfaces (GUI). It's implementation would typically delegate the tasks by calling methods of interfaces *AttrManager* and *AttrEditorManager* (the latter can be found in the package agg.attribute.gui. Figure 6.4 displays the methods of the *EditorFacade* interface. It also shows the interfaces implicitly involved.

The detailed description of this facade's methods can be found in the appendix. Here an example of using this facade interface:

```
/////////////////////////////////////////////////////
// Example of using the 'EditorFacade' interface:
/////////////////////////////////////////////////////

import agg.attribute.*;
import agg.attribute.gui.*;
import agg.attribute.facade.*;
import agg.attribute.facade.impl.DefaultEditorFacade;
import java.awt.*;

// Getting the handle for the editor facade:
EditorFacade editorFacade = DefaultEditorFacade.self();

// Calling a compact editor for an attribute tuple instance;
// its component (window) can be incorporated into e.g. a node
// representation:

AttrInstance tupleInstance;
Container someGUI;
// ...
AttrTupleEditor instEditor =
    editorFacade.getSmallEditorForInstance( tupleInstance );

// Integrating the tuple editor into another GUI:
someGUI.add( instEditor.getComponent());

// Calling a comprehensive editor that allows editing of:
// - an attribute tuple, including its type member properties, such as
//   the member handlers, types and names; the hiding and moving of
//   members is possible;
```

```
// - a rule or match context, i.e. its variables and application
//   conditions;
// - options of the attribute component (customization).
// It can be integrated into a bigger application GUI panel:

AttrContext attrContext;
Container someBigGUI;
// ...
AttrTopEditor topEditor =
    editorFacade.getTopEditor();

// Loading the attribute instance tuple to edit:
editorFacade.editInstance( topEditor, tupleInstance );

// Loading the attribute context to edit:
editorFacade.editInstance( topEditor, attrContext );

// Integrating the top editor into another GUI:
someBigGUI.add( topEditor.getComponent());

// End of example
//////////////////////////////////////////////////////////
```

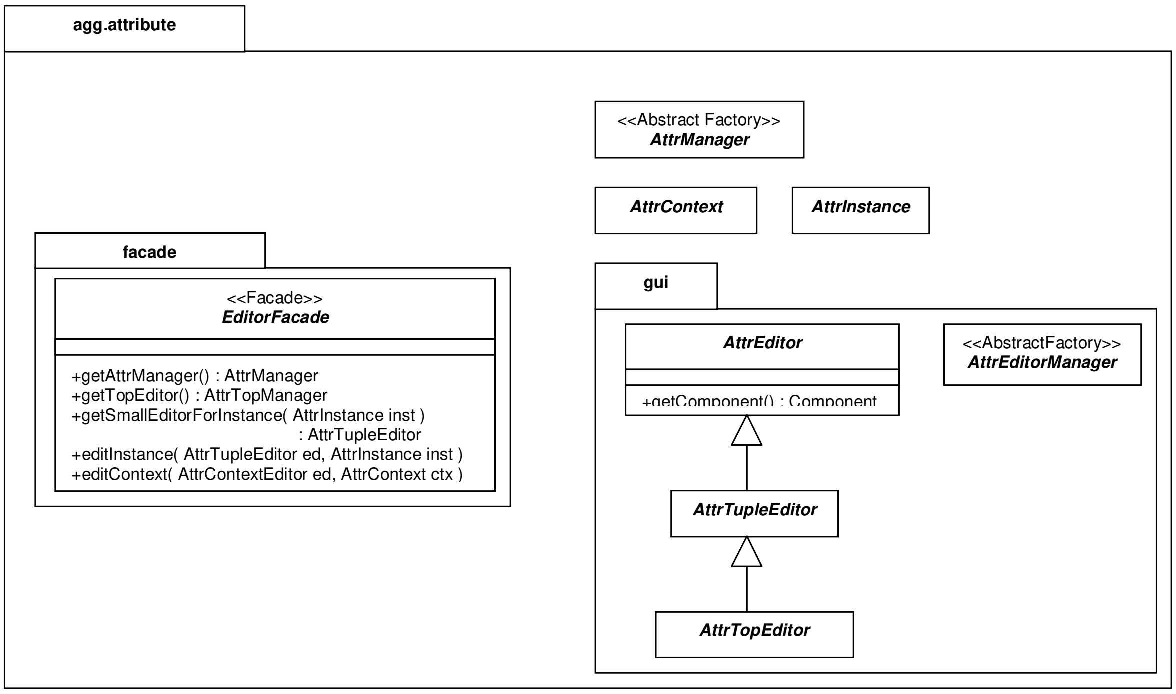

Figure 6.4: Editor Facade

6.4 Distributed Graph Transformation

This section shows an example of how to employ the attribute component in a distributed graph transformation environment. The distributed aspects are used combining services of *agg.attribute.facade.InformationFacade*, *agg.attribute.facade.TransformationFacade* and *agg.attribute.AttrDistributionBroker*. The example below follows that in section 2.3.2, see figures 2.14 on page 31 and 2.15 on page 32.

```
//////////////////////////////////////////////////////
// Example of using distribution:
//////////////////////////////////////////////////////

import agg.attribute.*;
import agg.attribute.facade.*;
import agg.attribute.facade.impl.DefaultInformationFacade;
import agg.attribute.facade.impl.DefaultTransformationFacade;

// Getting the handles for the information and the transformation
// facades:

InformationFacade infoFacade = DefaultInformationFacade.self();
TransformationFacade transFacade = DefaultTransformationFacade.self();

// Getting the handle for the distribution broker:

AttrDistributionBroker broker =
infoFacade.getAttrManager().getDistributionBroker();

// Creating new attribute tuple types and establishing
// their distribution relationship:

AttrType interfaceType_Person = infoFacade.createTupleType();
AttrType localType_Client = infoFacade.createTupleType();
AttrType localType_Student = infoFacade.createTupleType();

broker.connect( interfaceType_Person, localType_Client );
broker.connect( interfaceType_Person, localType_Student );

  // ... filling the types with information ...

// Creating new rule contexts for the interface and the local
// rule parts:
```

```
AttrContext interfaceRuleContext = transFacade.newRuleContext();
AttrContext localRuleContext_Library = transFacade.newRuleContext();
AttrContext localRuleContext_School = transFacade.newRuleContext();

broker.connect( interfaceRuleContext, localRuleContext_Library );
broker.connect( interfaceRuleContext, localRuleContext_School );

// Creating contexts for their left and right rule sides:

AttrContext interfaceLeftContext =
transFacade.getLeftContext( interfaceRuleContext );
AttrContext interfaceRightContext =
transFacade.getRightContext( interfaceRuleContext );

AttrContext localLeftContext_Library =
transFacade.getLeftContext( localRuleContext_Library );
AttrContext localRightContext_Library =
transFacade.getRightContext( localRuleContext_Library );
broker.connect( interfaceLeftContext, localLeftContext_Library );
broker.connect( interfaceRightContext, localRightContext_Library );

  // ... similar for the "school" rule contexts ...

// Creating contexts for the negative application conditions (NACs):

AttrContext interfaceNACRuleContext =
transFacade.getNegativeRuleContext( interfaceRuleContext );

AttrContext localNACRuleContext_Library =
transFacade.getNegativeRuleContext( localRuleContext_Library );
broker.connect( interfaceNACRuleContext, localNACRuleContext_Library );

  // ... similar for the "school" NAC rule contexts ...

//     manipulating their values, editing variables and application
//     conditions of the rule context ...

// Creating distributed attribute tuple instances on both rule sides,
// in the NAC and in the work graph:

// Left rule side:

AttrInstance interfaceLeft_Person =
```

```
infoFacade.createTupleInstance( interfaceType_Person,
interfaceLeftContext );

AttrInstance localLeft_Client =
infoFacade.createTupleInstance( localType_Client,
localLeftContext_Library );
broker.connect( interfaceLeft_Person, localLeft_Student );

AttrInstance localLeft_Student =
infoFacade.createTupleInstance( localType_Student,
localLeftContext_School );
broker.connect( interfaceLeft_Person, localLeft_Client );

// Right rule side:

AttrInstance interfaceRight_Person =
infoFacade.createTupleInstance( interfaceType_Person,
interfaceLeftContext );
AttrInstance localRight_Client =
infoFacade.createTupleInstance( localType_Client,
broker.connect( interfaceRight_Person, localRight_Client );
localRightContext_Library );
AttrInstance localRight_Student =
infoFacade.createTupleInstance( localType_Student,
localRightContext_School );
broker.connect( interfaceRight_Person, localRight_Student );

// Negative application condition (NAC) graph:

AttrInstance interfaceNAC_Person =
infoFacade.createTupleInstance( interfaceType_Person,
interfaceNACContext );
AttrInstance localNAC_Client =
infoFacade.createTupleInstance( localType_Client,
localNACContext_Library );
broker.connect( interfaceNAC_Person, localNAC_Client );

AttrInstance localNAC_Student =
infoFacade.createTupleInstance( localType_Student,
localNACContext_School );
broker.connect( interfaceNAC_Person, localNAC_Student );

// Work graph:
```

```
AttrInstance interfaceWork_Person =
infoFacade.createTupleInstance( interfaceType_Person, null );

AttrInstance localWork_Client =
infoFacade.createTupleInstance( localType_Client, null );
broker.connect( interfaceWork_Person, localWork_Client );

AttrInstance localWork_Student =
infoFacade.createTupleInstance( localType_Student, null );
broker.connect( interfaceWork_Person, localType_Student );

// Mapping left side attribute tuple instances to those
// on the right side within the rule contexts must be explicitly
// invoked for the interface and each of its local systems, it is NOT
// automatically invoked for the local instances when mapping interface
// instances.
// ...

// Mapping left side attribute tuple instances to those
// in the negative application condition must be explicitly
// invoked for the interface and each of its local systems, it is NOT
// automatically invoked for the local instances when mapping interface
// instances.
// ...

// When an attribute mapping is to be dissolved, this has to be done
// for the interface and each of its local systems.
// ...

// Checking, if the rule contexts are consistent, i.e. there are no
// contradictory equations in the application conditions:

try{
    transFacade.checkIfReadyToMatch( interfaceRuleContext );
    transFacade.checkIfReadyToMatch( localRuleContext_Library );
    transFacade.checkIfReadyToMatch( localRuleContext_School );
}
catch( AttrException ex ){
    System.out.println( ex.getMessage());
    return;
}
```

```
// Creating match contexts:

AttrContext interfaceMatchContext =
transFacade.newMatchContext( interfaceRuleContext );

AttrContext localMatchContext_Library =
transFacade.newMatchContext( localRuleContext_Library );
broker.connect( interfaceMatchContext, localMatchContext_Library );

AttrContext localMatchContext_School =
transFacade.newMatchContext( localRuleContext_School );
broker.connect( interfaceMatchContext, localMatchContext_School );

// Mapping left side tuple instances to those in the work graph,
// within the match context. Hereby, an attempted mapping can fail,
// whereupon the cause can be analyzed by the match searching algorithm:

AttrMapping matchMapping;

try{
    matchMapping =
        transFacade.newMapping( interfaceMatchContext,
interfaceLeft_Person,
interfaceWork_Person );
    matchMapping =
        transFacade.newMapping( localMatchContext_Library,
localLeft_Client,
localWork_Client );
    matchMapping =
        transFacade.newMapping( localMatchContext_School,
localLeft_Student,
localWork_Student );
}
catch( AttrMatchException ex ){
    int id = ex.getID();
    if( id == AttrMatchException.VARIABLE_BINDING ){
        AttrInstance firstBindingTuple = ex.getFirstBindingTuple();
        // ...
    }
}

// Creating negative match contexts:
```

```
AttrContext negMatchContext =
transFacade.newNegativeMatchContext( negativeRuleContext0,
     matchContext );

AttrContext interfaceNACMatchContext =
transFacade.newNegativeMatchContext( interfaceNACRuleContext,
     interfaceMatchContext );

AttrContext localNACMatchContext_Library =
transFacade.newNegativeMatchContext( localNACRuleContext_Library,
     localMatchContext_Library );
broker.connect( interfaceNACMatchContext, localNACMatchContext_Library );

AttrContext localNACMatchContext_School =
transFacade.newNegativeMatchContext( localNACRuleContext_School,
     localMatchContext_School );
broker.connect( interfaceNACMatchContext, localNACMatchContext_School );

// Checking, if the match context is consistent and transformation
// of attributes can be performed:

try{
    transFacade.checkIfReadyToTransform( interfaceMatchContext );
    transFacade.checkIfReadyToTransform( localMatchContext_Library );
    transFacade.checkIfReadyToTransform( localMatchContext_School );
}
catch( AttrException ex ){
    System.out.println( ex.getMessage());
    return;
}

// Performing attribute transformation:

transFacade.apply( interfaceWork_Person, interfaceRight_Person,
   interfaceMatchContext );

transFacade.apply( localWork_Client, localRight_Client,
   localMatchContext_Library );

transFacade.apply( localWork_Student, localRight_Student,
   localMatchContext_School );

// End of example
```

//

Chapter 7

System Implementation

This chapter describes the system implementation and addresses developers who wish to maintain and extend the existing system. The main focus will be:

- the structure of implementing classes and how it parallels the interface structure;
- the event-propagation mechanism and subject/observer dependencies within the implementation.

First, the tuple implementation structure is explained, along with the class inheritance hierarchy. This includes attribute declarations, values and context variables and conditions. Then, the attribute context, observer mechanism, the views and the editor hierarchy are presented. After giving an overview of the Java interpreter classes, the test environment for the attribute component is described.

7.1 Attribute Tuples and Their Members

Upon presenting an illustrating example of tuple types and instances, an overview of the tuple implementation structure is given. Then, all the tuple implementations (declarations, values, context variables and conditions) are described.

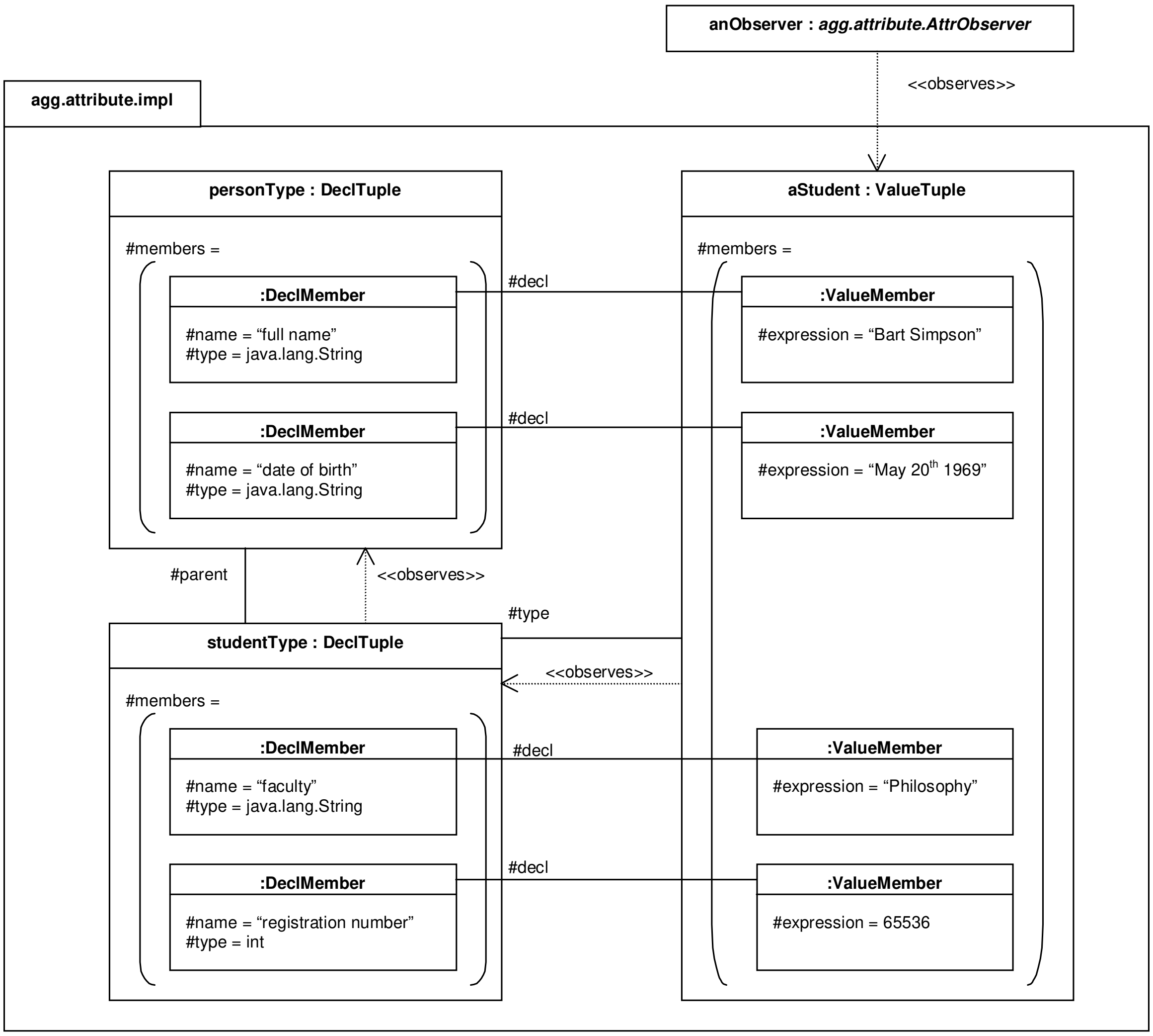

Figure 7.1: Example instance diagram with tuple/members-relationships

7.1.1 Example Object Diagram

The figure 7.1 is an example instance diagram showing dependencies in relationships between attribute tuples and members. It illustrates some essential implementation requirements when it comes to attribute tuples. In the presented example, there is the attribute tuple type *Person* with the member declarations *full name* and *date of birth*. This type is extended (or sub-typed) by the attribute tuple type *Student* that adds the member declarations *faculty* and *registration number*. There is also an attribute tuple instance *aStudent* with certain value expressions.

The inheritance mechanism is supported as indicated, by using a `parent` association. A child tuple is an observer of its parent tuple, a value tuple is an observer of its type (declaration tuple). The observer relations will be explained in more detail in section 7.3.

7.1.2 Tuple Implementation Overview

Figure 7.2 shows the implementation architecture of the tuple structure. It is an overview of how the interfaces in figure 5.1 on page 53 are implemented. The diagram shows how the implementing structure parallels the interface structure in figure 5.1 on page 53, figure 5.9 on page 68 and figure 5.10 on page 70. The exception is the class LoneTuple that has no corresponding interface. It implements the property of a value tuple whose declaration tuple has no other value tuples. This property is shared by the variable tuple and the condition tuple.

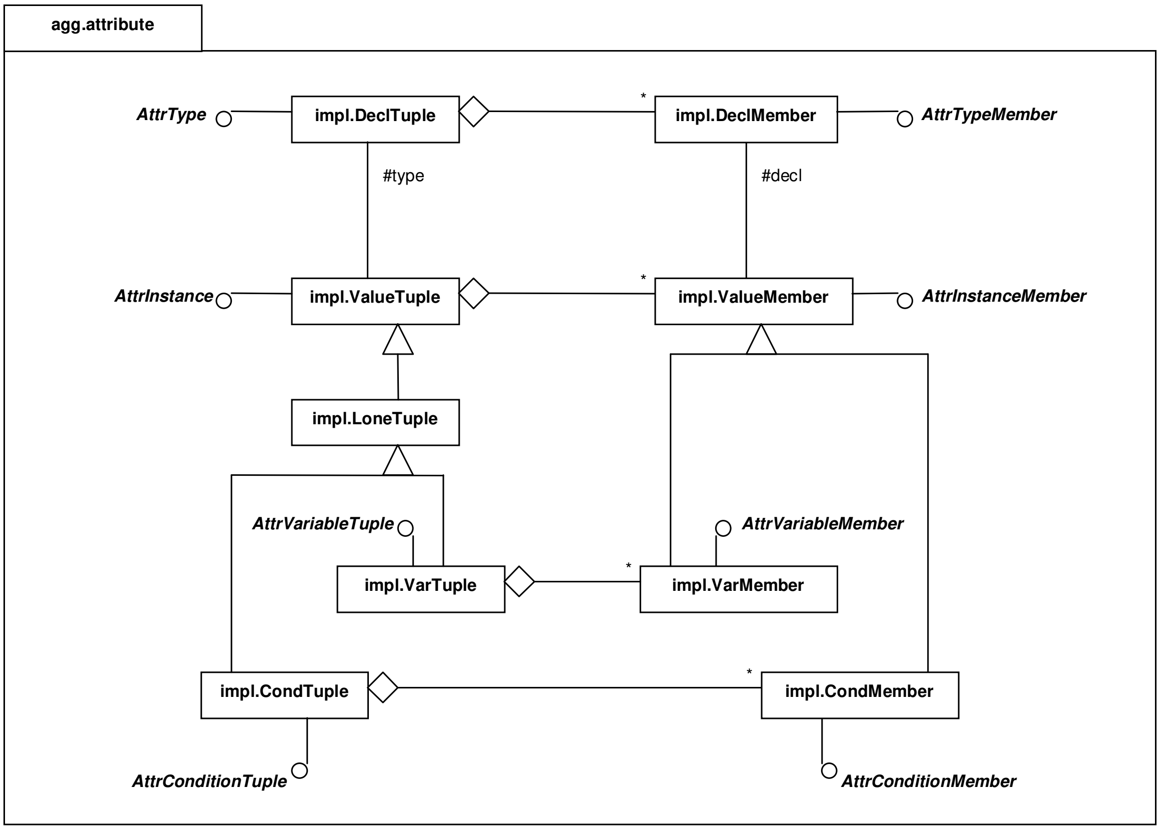

Figure 7.2: Implementation architecture of the tuple structure

7.1.3 TupleObject - The Root Class for all Tuples

Figure 7.3 shows the implementation of the TupleObject class. The explanations of its members and the associated classes follow below.

AttrTuple, AttrMember, AttrManager These interfaces are described in section 5.2.1 on page 50ff.

TupleObject Partial implementation of the interface agg.attribute.AttrTuple. Adds the abstract method getTupleType().

- protected TupleObject parent = null Parent of this type. All parent entries are "inherited".
- protected Vector members = new Vector(10, 10) Container with members, all of which implement the AttrMember interface.
- public TupleObject(AttrTupleManager manager, TupleObject parent) Constructor. Parameter 'parent' can be null.
- protected void assignParent(TupleObject newParent) Setting the parent tuple.
- protected TupleObject getParent() Returns the parent tuple.
- protected int getParentSize() Obtaining the size of the parent tuple.
- protected TupleObject getParentInCharge(int index) Returns the ancestor that originated the member at 'index'.
- public boolean isSubclassOf(TupleObject maybeParent) Returns 'true' if this tuple is a descendant of 'maybeParent', 'false' otherwise.
- public abstract DeclTuple getTupleType() This method interface is needed in order to treat attribute types and instances uniformly.
- public AttrType getType() Returns the type of the tuple.

The following methods with the name prefix 'raw' provide container-specific access to the container of [members] (currently: java.util.Vector). Singling these methods out allows for easy replacement of the container, e.g. by a more efficient one.

- protected synchronized int rawGetSize() Returns the number of members in the underlying container.
- protected synchronized AttrMember rawGetMemberAt(int index) Returns the member at 'index' in the underlying container.
- protected synchronized void rawAddMember(AttrMember member) Adds a member to the underlying container.
- protected synchronized void rawDeleteMemberAt(int index) Removes a member from the underlying container.

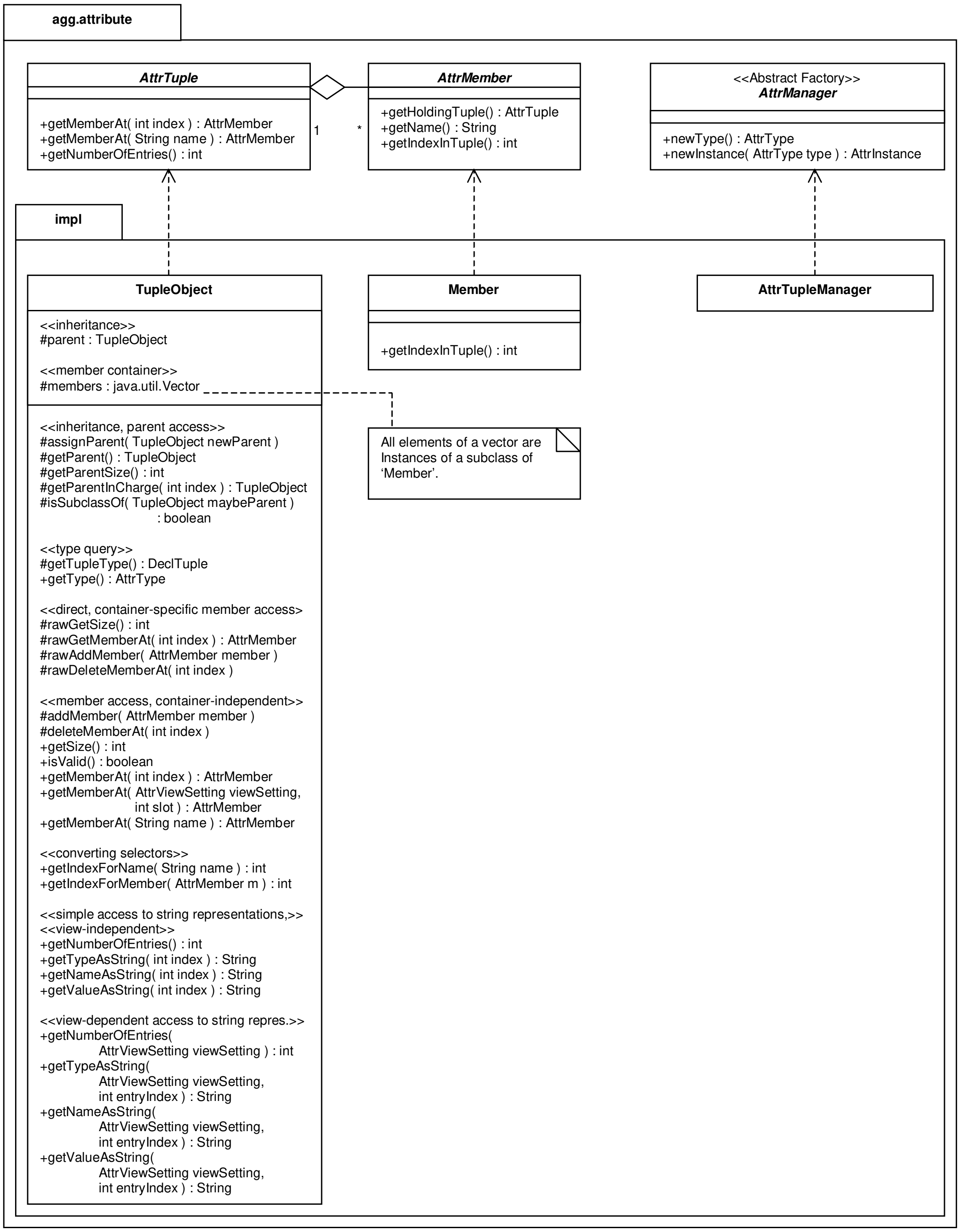

Figure 7.3: TupleObject and Member, the root classes for all tuples and members

The following methods provide container-independent access to the state and the members of the tuple. They delegate access to containers to the previous, 'raw' mathods.

- protected void addMember(AttrMember member) Adding a new member to the tuple.
- protected void deleteMemberAt(int index) Removing a member from the tuple.
- public int getSize() Returns the number of members in the tuple.
- public boolean isValid() Returns 'true' if all the tuple's members are valid, 'false' otherwise.
- public AttrMember getMemberAt(int index) Getting a tuple member by its absolute (view-independent) index.
- public AttrMember getMemberAt(AttrViewSetting viewSetting, int slot) Getting a tuple member by its view-dependent index.
- public AttrMember getMemberAt(String name) Getting a tuple member by its declaration name.

Converting selectors:

- public int getIndexForName(String name) Translation between number- and name-oriented access. Returns the corresponding index if the name is declared within the tuple, -1 otherwise.
- public int getIndexForMember(AttrMember m) Translation between address- and number-oriented access. Return the corresponding index if the member is within the tuple, -1 otherwise.

Simple access to textual representations, view-independent:

- public int getNumberOfEntries() Getting the total number of shown attribute entries (lines); The retrieval index range is [0 .. (getNumberOfEntries() - 1)].
- public String getTypeAsString(int index) Getting a simple representation of a type as String.
- public String getNameAsString(int index) Getting a simple representation of a name as String.
- public String getValueAsString(int index) Getting a simple representation of a value as String.

Simple access to textual representations, view-dependent:

- public int getNumberOfEntries(AttrViewSetting viewSetting) Getting the view-dependent number of attribute entries (lines). The retrieval index range is [0 .. (getNumberOfEntries() - 1)].
- public String getTypeAsString(AttrViewSetting viewSetting, int entryIndex) Getting a view-dependent representation of a type as String.
- public String getNameAsString(AttrViewSetting viewSetting, int entryIndex) Getting a view-dependent representation of a name as String.
- public String getValueAsString(AttrViewSetting viewSetting, int entryIndex) Getting a view-dependent representation of a value as String.

Member Partial implementation of the 'AttrMember' interface. This class is a parent class of all member classes: DeclMember, ValueMember, VarMember, CondMember.

- public int getIndexInTuple() Returns the index for this member in its tuple.

AttrTupleManager Attribute Tuple Manager - Factory of attribute management; provides creating services needed by graph components.

7.1.4 DeclTuple - Tuple of Declaration Members

Figure 7.4 shows the implementation of the DeclTuple class. The explanations of its members and the associated classes follow below.

AttrTuple, AttrMember, AttrType, AttrTypeMember, AttrManager These interfaces are described in section 5.2.1 on page 50ff.

TupleObject, Member, AttrTupleManager These classes are described in section 7.1.3 on page 116ff.

DeclTuple Implements agg.attribute.AttrTuple. Each attribute tuple instance has exactly one type, which is realized by this class.

- protected AttrViewSetting fixedFormSetting The view setting which is used for changing the order of members for internal access.

Public Constructors:

- public DeclTuple(AttrTupleManager manager) Creates a new DeclTuple with no parent and the specified attribute manager.
- public DeclTuple(AttrTupleManager manager, DeclTuple parent) Creates a new DeclTuple with the specified parent and attribute manager.

Overriding size query:

- public int getSize() Own size is added to the parents' size, recursively, so the result ist the total number of declarations known to this instance.

Conversions between leaf and root indices; leaf index relates to this declaration tuple itself, root index relates to the root parent of this tuple:

- protected int toLeafIndex(int rootIndex) Transforming a root index into the corresponding index of this leaf.
- protected int toRootIndex(int leafIndex) Obtaining the index relative to the root of the inheritance tree.
- protected AttrMember getLeafMemberAt(int rootIndex) Getting a member from the container of this child.

Convenience method for member access:

- protected DeclMember getDeclMemberAt(int index) Works much like the generic getMemberAt(int index), but returns the appropriate member type.

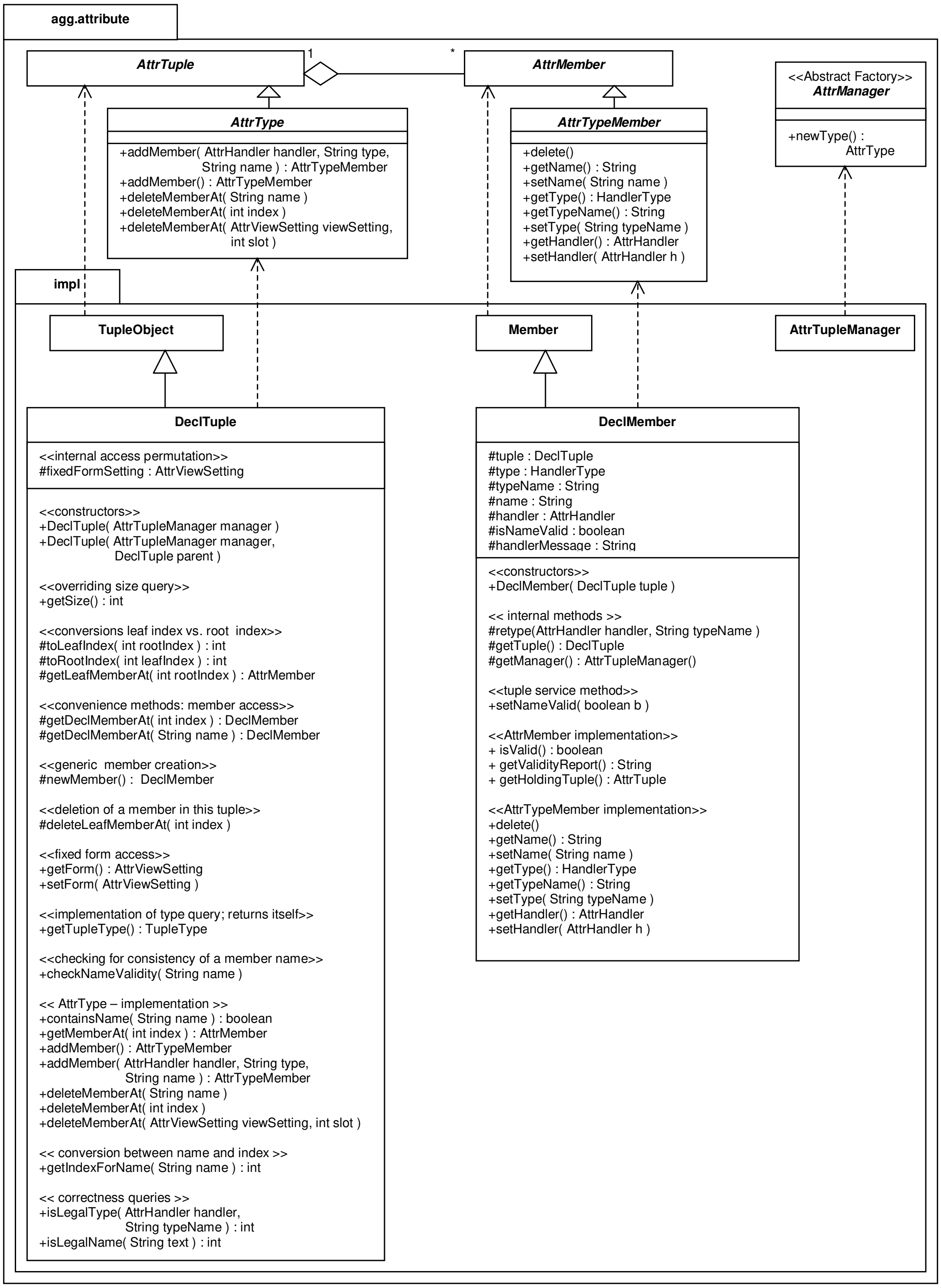

Figure 7.4: DeclTuple - the tuple of declaration members

- protected DeclMember getDeclMemberAt(String name) Works much like the generic getMemberAt(String name), but returns the appropriate member type.

Generic member creation:

- protected DeclMember newMember() Subclasses use/override this method in order to create their own members of the appropriate type.

Deletion of a member in this tuple:

- protected void deleteLeafMemberAt(int rootIndex) Deletes the member of this tuple, where the 'rootIndex' points to a member in this tuple part (leaf).

Fixed form access; a 'fixed form' is a special, internal view setting that not only determines the tuple layout but also the order of processing the tuple members. No hiding is involved, just permutation. Condition tuple (CondTuple) and variable tuple (VarTuple) use the fixed form so it is possible to change the order of condition checking or function values, respectively:

- protected AttrViewSetting getForm() Returns the view setting containing the permutation of internal access indexes.
- protected void setForm(AttrViewSetting formSetting) Sets the view setting containing the permutation of internal access indexes.

Implementation of type query; returns itself:

- public DeclTuple getTupleType() Returns the tuple type, i.e. this DeclTuple instance itself.

Service for DeclMember:

- public void checkNameValidity(String name) Checking the validity of a name as a unique key for a member. Tests if 'name' is assigned to more than one member. If so, the method 'setNameValid(false)' is invoked on every such member. Otherwise, 'setNameValid(true)' is invoked on the member with this name if such a member exists. Called by DeclMember when changing the name.

AttrType - implementation:

- public boolean containsName(String name) Querying if an entry is declared.
- public AttrMember getMemberAt(int index) Returns the member at a particular index. Overrides getMemberAt() in TupleObject, taking the inheritance parts into consideration.

- public AttrTypeMember addMember() Adds a new member to this tuple. Overrides addMember() in TupleObject, creating the new member instance with the appropriate member type (DeclMember).
- public AttrTypeMember addMember(AttrHandler handler, String type, String name) Convenience method for adding a new member. Creates a member with the specified attribute handler, expression type and selector name.
- public void deleteMemberAt(int index) Deletes the member at a particular index. Overrides deleteMemberAt() in TupleObject, taking the inheritance parts into consideration.
- public void deleteMemberAt(AttrViewSetting viewSetting, int slot) Deletes the member at a particular slot, relative to the specified view setting.
- public void deleteMemberAt(String name) Deletes the member with a certain name.

Conversion between name and index:

- public int getIndexForName(String name) Translation between number- and name-oriented access. Returns the corresponding index if the name is declared within the tuple, -1 otherwise.

Correctness queries:

- public int isLegalType(AttrHandler handler, String typeName) Check if 'handler' says that it can make a type out of 'typeName'.
- public int isLegalName(String text) Check if 'text' not already defined as name.

DeclMember Keeps the declaration name, type and the type's handler.

Instance variables:

- protected DeclTuple tuple The tuple containing this declaration.
- protected HandlerType type The type of this declaration.
- protected String typeName Type name.
- protected String name The name of this declaration.
- protected AttrHandler handler The attribute handler that created the type.
- protected boolean isNameValid Flag if the member name is unique within its tuple.
- protected String handlerMessage Last error message from the attribute handler.

Constructor:

- public DeclMember(DeclTuple tuple) Constructing for a tuple.

Internal methods:

- protected void retype(AttrHandler handler, String typeName) Changing the declaration type.
- protected DeclTuple getTuple() Returns the tuple that contains this member.
- protected AttrTupleManager getManager() Returns the attribute manager for this member's tuple.

Tuple service method:

- public void setNameValid(boolean b) Setting if the name is valid (unique in the tuple). Called by DeclTuple.

AttrMember - implementation:

- public boolean isValid() Returns if this declaration member is valid. Returns 'true' if the name is unique within the tuple and the member has a type; 'false' otherwise.
- public String getValidityReport() If the member is valid, returns 'null'. Otherwise, returns a text describing the reason why the member is not valid.
- public AttrTuple getHoldingTuple() Returns the tuple that contains this member.

AttrTypeMember interface implementation:

- public void delete() Deletes this member from its tuple.
- public String getName() Retrieving its name.
- public void setName(String name) Setting a name.
- public HandlerType getType() Retrieving its type.
- public String getTypeName() Retrieving its type name as string.
- public void setType(String typeName) Setting its type.
- public AttrHandler getHandler() Retrieving its attribute handler.
- public void setHandler(AttrHandler h) Setting its attribute handler.

7.1.5 ValueTuple - Tuple of Value Members

Figure 7.5 shows the implementation of the ValueTuple class. The explanations of its members and the associated classes follow below.

AttrTuple, AttrMember, AttrInstance, AttrInstanceMember, AttrManager These interfaces are described in section 5.2.1 on page 50ff.

TupleObject, Member, AttrTupleManager These classes are described in section 7.1.3 on page 116ff.

ValueTuple Implementation of the interface agg.attribute.AttrInstance; Encapsulates a tuple of value member.

Protected instance variables:

- protected DeclTuple type Type reference.
- protected ContextView context Context in which this instance is placed.

Constructors:

- public ValueTuple(AttrTupleManager manager, DeclTuple type) Creates a new attribute value tuple with 'type' as declaration tuple and no context.
- public ValueTuple(AttrTupleManager manager, DeclTuple type, ContextView context) Creates a new attribute value tuple with a type and a context.
- public ValueTuple(AttrTupleManager manager, DeclTuple type, ContextView context, ValueTuple parent) Creates a new attribute value tuple with a type and a context. All entries in 'parent' are copied and 'parent' is observed. This was intended for the implementation of distribution, but distribution is now realized in a different way (using stubs), so this 'parent'-mechanism is never used with value tuples.

Internal Methods:

- protected void assignParent(TupleObject newParent) Assigning a parent value tuple; Adopts the new parent's entries.
- protected void setType(DeclTuple type) Setting a type.
- protected void adaptToType() Causes the value container (Vector) size to match the type container size.
- protected ValueMember newMember(DeclMember decl) Generic method for creation of a new value member.

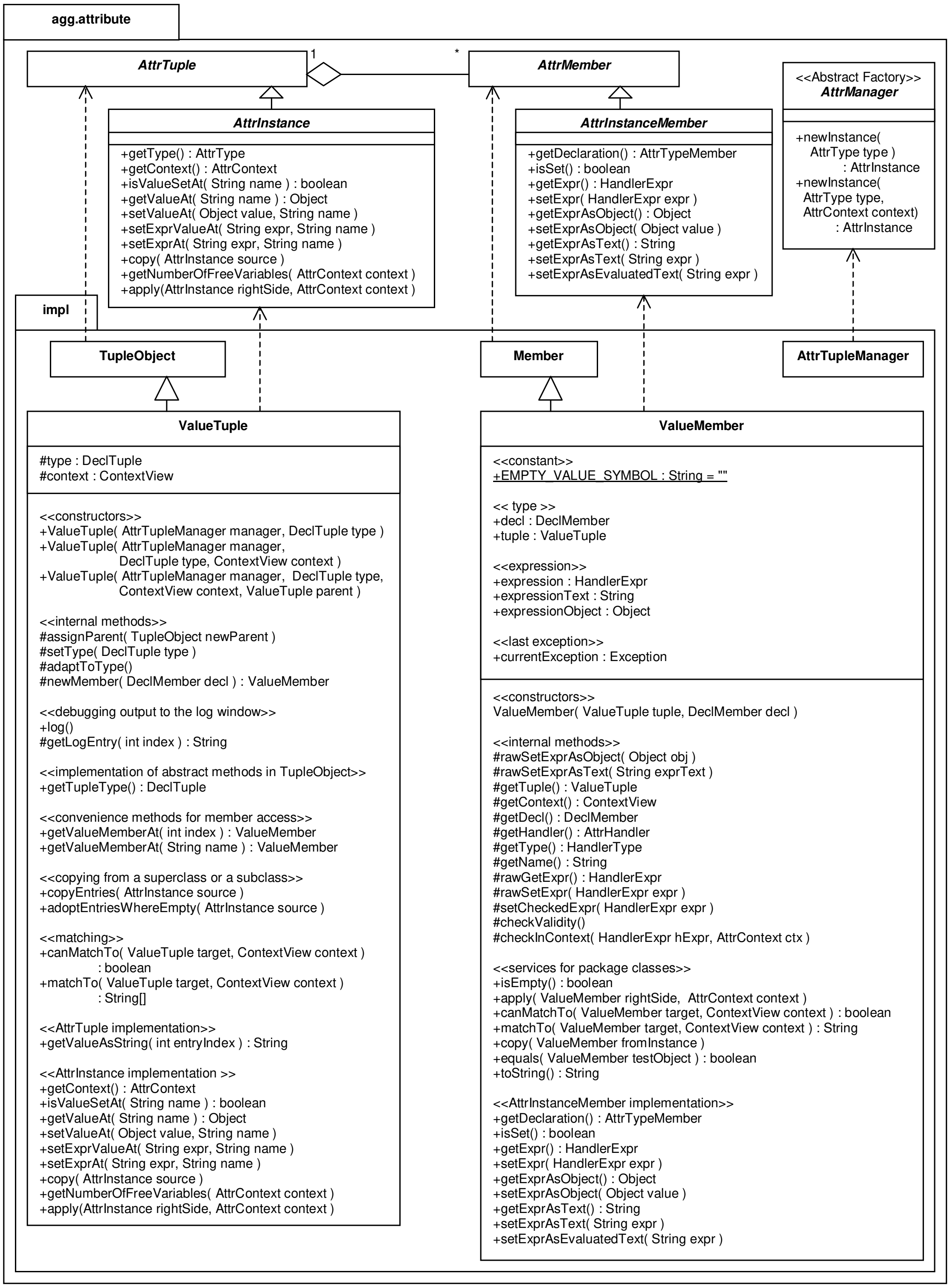

Figure 7.5: ValueTuple - the tuple of value members

Debugging output to the log window:

- public void log() For debugging: displaying itself in the logging window.
- protected String getLogEntry(int index) Sub-procedure for 'log()', creates a text showing the entry at 'index'.

Implementation of abstract methods in TupleObject:

- public DeclTuple getTupleType() Retrieving the type of this tuple instance.

Convenience methods for member access:

- public ValueMember getValueMemberAt(int index) Convenience method for internal operations; works much like the generic getMemberAt(int index), but returns the appropriate member type.
- public ValueMember getValueMemberAt(String name) Convenience method for internal operations; works much like the generic getMemberAt(String name), but returns the appropriate member type.

Copying from a super class or a subclass:

- public void copyEntries(AttrInstance source) Copying the contents of an attribute instance into another; 'source' must be of a supertype or a subtype of this value tuple's type.
- public void adoptEntriesWhereEmpty(AttrInstance source) The empty members of this value member are filled with values of corresponding members of 'source'. 'source' must be of a super type or a subtype of this value tuple's type.

Matching:

- public boolean canMatchTo(ValueTuple target, ContextView context) Querying if this value tuple matches the value tuple 'target' in the match context 'context'.
- public String[] matchTo(ValueTuple target, ContextView context) Performs a match in a per-element-style. Returns an array of the variable names that were assigned a value by this match.

AttrTuple implementation:

- public String getValueAsString(int entryIndex) Getting a simple representation of a value as String.

AttrInstance implementation:

- public AttrContext getContext() Returns the context of the value tuple.
- public boolean isValueSetAt(String name) Test, if a value is set or not.
- public Object getValueAt(String name) Retrieving the value of an entry.
- public void setValueAt(Object value, String name) Setting the value of an entry directly.
- public void setExprValueAt(String expr, String name) Evaluating an expression and setting its value as an entry.
- public void setExprAt(String expr, String name) Setting an expression as an entry without immediate evaluation. Syntax and type checking are performed.
- public void copy(AttrInstance source) Copying the contents of an attribute instance into another; The reference to the attribute type is shared.
- public int getNumberOfFreeVariables(AttrContext ctx) Getting the number of variables declared by this instance which have no value assigned to them yet. Each variable name is counted only once, even if it is used more than once in this tuple.
- public void apply(AttrInstance rightSide, AttrContext context) Applying a rule; the substitutions occur "in-place", i.e. in the recipient; In Graph Transformation, this method is applied to attributes of host graph objects, "rightSide" being an attribute of the right side of the rule and "context" being the "match"-context built up by subsequently matching the attributes of corresponding graphical objects.

ValueMember Holds an attribute handler expression, its type and the functionality for matching and transforming thereof.

Constant:

- static public final String EMPTY_VALUE_SYMBOL = "" This string is shown for an empty value.

Type:

- protected DeclMember decl Declaration.
- protected ValueTuple tuple Instance tuple that this value is a member of.

Attribute handler expression:

- protected HandlerExpr expression
- protected String expressionText
- protected Object expressionObject

Last exception:

- protected Exception currentException

Constructors:

- public ValueMember(ValueTuple tuple, DeclMember decl) Creating a new instance with the specified type.

Internal methods:

- protected void rawSetExprAsObject(Object obj) Low-level setting of the value using an object.
- protected void rawSetExprAsText(String exprText) Low-level setting of the value using a textual representation.
- protected ValueTuple getTuple() Getting the instance tuple that contains this value as a member.
- protected ContextView getContext() Getting the context of this value.
- protected DeclMember getDecl() Getting the declaration for this value.
- protected AttrHandler getHandler() Getting the handler of this value.
- protected HandlerType getType() Getting the type of this value.
- public String getName() Getting the name of this value member.
- protected HandlerExpr rawGetExpr() Low-level retrieval of the value as HandlerExpr instance.
- protected void rawSetExpr(HandlerExpr expr) Low-level setting of the value as HandlerExpr instance.
- protected void setCheckedExpr(HandlerExpr expr) Setting the value and checking validity.
- protected void checkValidity() Checking the current expression validity relative to my tuple's context.
- protected void checkInContext(HandlerExpr hExpr, AttrContext ctx) Checking the validity of the expression 'hExpr' relative to the context 'ctx'.

Services for package classes:

- public boolean isEmpty() Check if no expression yet.
- public void apply(ValueMember rightSide, AttrContext context) Transformation application.

- public boolean canMatchTo(ValueMember target, ContextView context) Check if matching is possible into 'target' within the match context 'context'.
- public String matchTo(ValueMember target, ContextView context) Perform matching with 'target' in the match context 'context'.
- public void copy(ValueMember fromInstance) Copying the contents of a single entry instance into another.
- public boolean equals(ValueMember testObject) Tests if the handler expressions are equal.
- public String toString() Returns the textual representation of the expression or 'EMPTY_VALUE_SYMBOL' if that is null.

The methods of AttrInstanceMember implementation are described in section 5.2.1 on page 51.

7.1.6 VarTuple - Tuple of Context Variables

Figure 7.6 shows the implementation of the VarTuple class. The explanations of its members and the associated classes follow below.

AttrTuple, AttrMember, AttrInstance, AttrInstanceMember These interfaces are described in section 5.2.1 on page 50ff.

AttrVariableTuple, AttrVariableMember These interfaces are described in section 5.4.3 on page 67ff.

TupleObject, Member These classes are described in section 7.1.3 on page 116ff.

ValueTuple, ValueMember These classes are described in section 7.1.5 on page 125ff.

LoneTuple A tuple whose type is not shared by others. Provides the possibility of reordering of its members using setForm(manager.getFixedViewSetting()), implemented in TupleObject. To be extended by VarTuple and CondTuple.

Constructor:

- public LoneTuple(AttrTupleManager manager, ContextView context, ValueTuple parent)

VarTuple Adds the possibility of being shared. Needed as the container of variable values inside a context core.

Constant:

- protected final int FIXED_VALUE = -1 A special value designating that the assignment was done in the parent context and this context's references don't count; such a value is permanent, it may not be changed by this value tuple.

Constructor:

- public VarTuple(AttrTupleManager manager, ContextView context, ValueTuple parent)

Internal Methods:

- protected ValueMember newMember(DeclMember decl) Generic method for creation of a new value, here: variable tuple member.

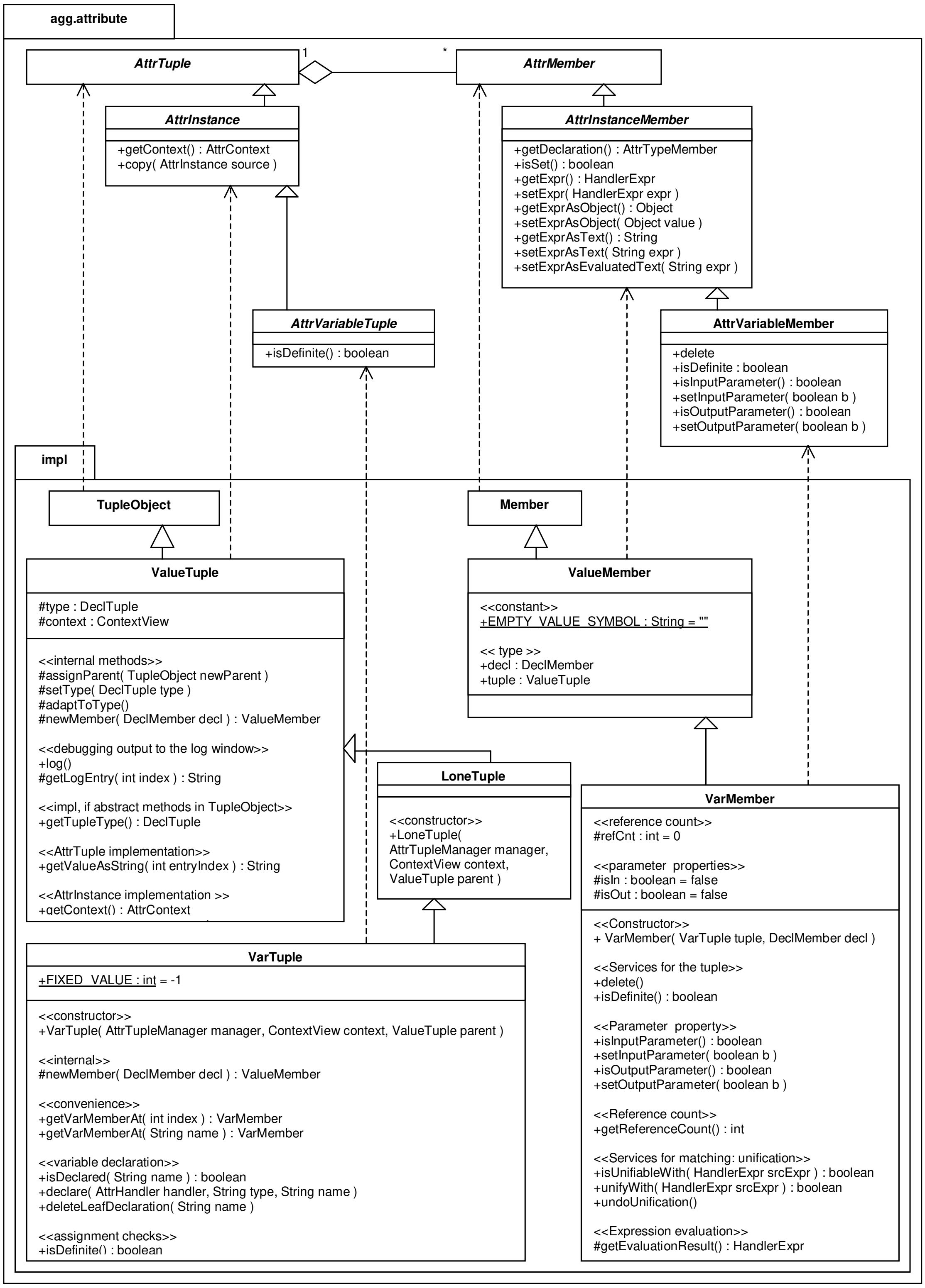

Figure 7.6: VarTuple - the tuple of context variables

Convenience:

- public VarMember getVarMemberAt(int index) Convenience method for internal operations; works much like the generic getMemberAt(int index), but returns the appropriate member type.
- public VarMember getVarMemberAt(String name) Convenience method for internal operations; works much like the generic getMemberAt(String name), but returns the appropriate member type.

Variable declaration:

- public void declare(AttrHandler handler, String type, String name) Convenience method for declaring and adding a new member. Creates a member with the specified attribute handler, expression type and selector name.
- public void deleteLeafDeclaration(String name) Deletes the variable with a certain name if it is declared in this tuple. If it is declared in a parent tuple, an AttrImplException is thrown.
- public boolean isDeclared(String name)

Returns 'true' if 'name' is already declared in the tuple, 'false' otherwise.

Assignment checks:

- public boolean isDefinite() Test, if all variables evaluate to definite values.

VarMember Class for members of attribute instance tuples that are used as variables in a context. This is an extension of ValueMember; it adds reference counting.

Variables:

- protected int refCnt = 0 Reference count.
- protected boolean isIn = false Flag, if the variable is an in-parameter.
- protected boolean isOut = false Flag, if the variable is an out-parameter.

Constructor:

- public VarMember(VarTuple tuple, DeclMember decl) Creating a new instance with the specified declaration.

Services for the tuple:

- public void delete() Removes this member from its tuple.
- public boolean isDefinite() Test, if the expression evaluates to a constant.

Parameter property:

- public boolean isInputParameter() Tests if this variable is an IN-parameter.
- public void setInputParameter(boolean in) Sets, if the variable is to be an IN-parameter.
- public boolean isOutputParameter() Tests if this variable is an OUT-parameter.
- public void setOutputParameter(boolean out) Sets, if the variable is to be an OUT-parameter.

Reference count:

- public int getReferenceCount() Returns the number of references on this variable.

Services for matching - unification:

- public boolean isUnifiableWith(HandlerExpr srcExpr) Returns 'true' if this variable is as yet undefined or if it has the same value as 'srcExpr', otherwise 'false'.
- public boolean unifyWith(HandlerExpr srcExpr) If this variable is unifiable with 'srcExpr', increments the reference count and returns 'true' (if this variable was not assigned a value yet, 'srcExpr' is assigned). Otherwise, returns 'false'.
- public void undoUnification() Cancels a unification; the reference count is decremented and if zero, this variable's value is set to 'null'.

Expression evaluation:

- protected HandlerExpr getEvaluationResult() Returns the result of evaluating this variable's expression; leaves this variable's expression unchanged.

7.1.7 CondTuple - Tuple of Context Conditions

Figure 7.7 shows the implementation of the CondTuple class. The explanations of its members and the associated classes follow below.

AttrTuple, AttrMember, AttrInstance, AttrInstanceMember These interfaces are described in section 5.2.1 on page 50ff.

AttrConditionTuple, AttrConditionMember These interfaces are described in section 5.4.4 on page 69ff.

TupleObject, Member These classes are described in section 7.1.3 on page 116ff.

ValueTuple, ValueMember These classes are described in section 7.1.5 on page 125ff.

LoneTuple This class is described in the previous section.

CondTuple Application conditions. Every instance of ContextCore has exactly one instance of this class.

Variables and constants:

- protected static String boolHandlerName = JexHandler.getLabelName() Name of the handler for the boolean type.
- final protected static String boolTypeName = "boolean" Constant for the boolean type name.
- final protected static String trueVal = "true" Constant for the true value.
- final protected static String falseVal = "false" Constant for the false value.
- protected int condNum = 0 Current condition number, is used to compose unique names within one condition tuple.

Constructor:

- public CondTuple(AttrTupleManager manager, ContextView context, CondTuple parent)

Internal:

- protected void initClass() Internal initialization.
- protected String getNextName() Generates a new, unique member name for this tuple.
- protected String getNameFor(int index) Determines the name for a certain index.

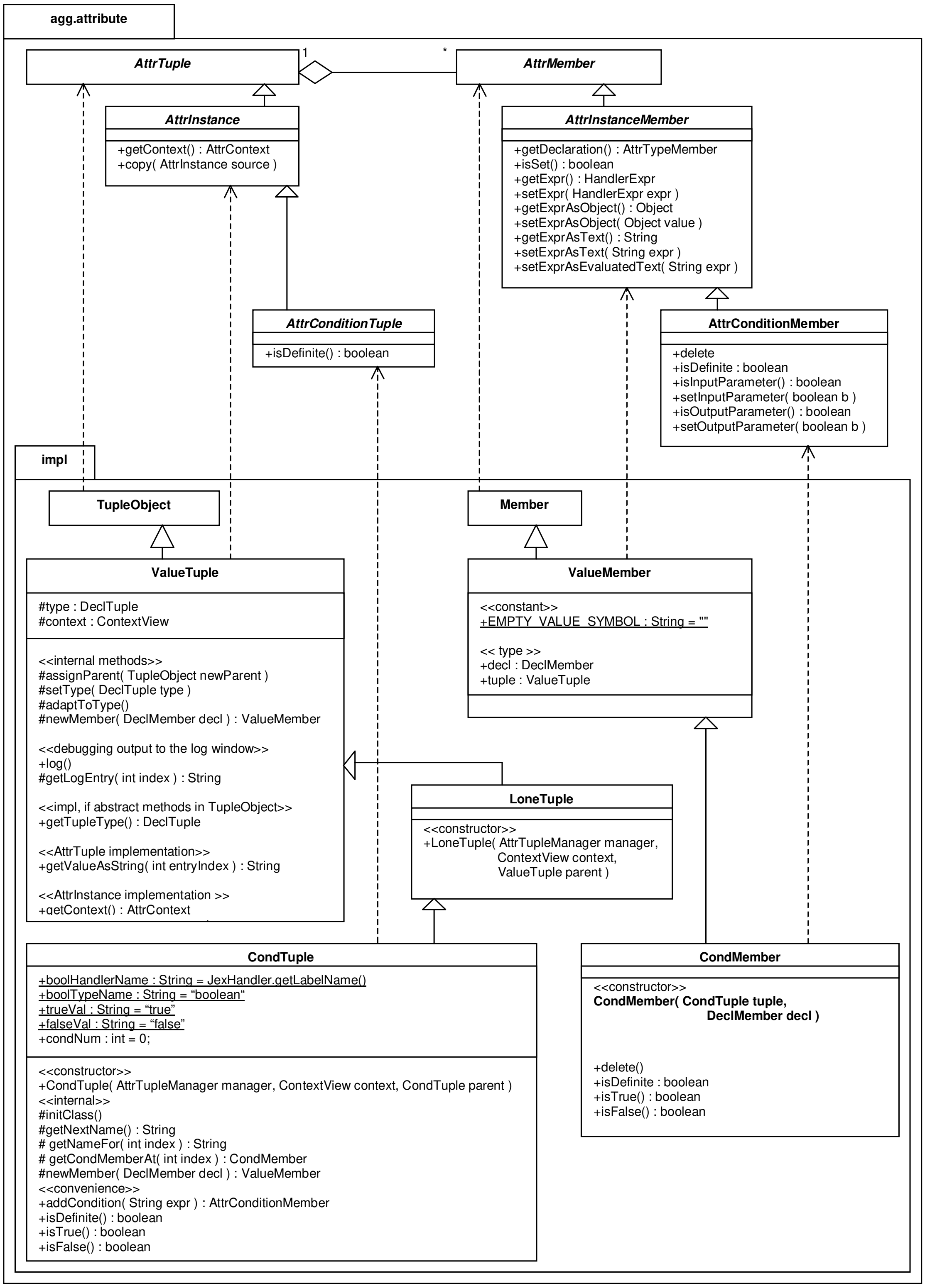

Figure 7.7: CondTuple - the tuple of context conditions

- protected CondMember getCondMemberAt(int index) Convenience method for internal operations; works much like the generic getMemberAt(String name), but returns the appropriate member type.
- protected ValueMember newMember(DeclMember decl) Generic component creation.

AttrConditionTuple - implementation:

- public AttrConditionMember addCondition(String expr) Adding of a condition member, returning the member.
- public boolean isDefinite() Test, if all members can yield true or false.
- public boolean isTrue() Test, if ANDing of all members yields true.
- public boolean isFalse() Test, if the tuple contains members which can be evaluated and yield 'false'.

CondMember Class for members of condition tuples that are used as application conditions in a context. This is an extension of ValueMember.

Constructor:

- public CondMember(CondTuple tuple, DeclMember decl) Creating a new instance with the specified type.

Services for condition tuples:

- public void delete() Removes this member from its tuple.
- public boolean isDefinite() Test, if the expression can yield true or false.
- public boolean isTrue() Test, if the expression yields true.
- public boolean isFalse() Test, if the expression yields false.

7.2 Attribute Context

Figure 7.8 shows the implementation of the context mechanism. The explanations of the participating classes follow below.

AttrContext, AttrVariableTuple, AttrConditionTuple, AttrMapping These interfaces are described in section 5.2.1 on page 50ff.

CondTuple, VarTuple These classes are described in the previous sections.

ContextView This is a view onto an underlying ContextCore class object; By this delegation, views with different access rights can share the same context; At this stage, two comprehensive access modes are implemented: "LeftRuleSide" and "RightRuleSide"; "RightRuleSide" access does not allow adding or removing of variable declarations; Finer access restrictions can be turned on and off using the 'setAllow...(boolean isAllowed)' - methods.

Describing the access mode:

- protected boolean canDeclareVar = true
- protected boolean canUseComplexExpr = true
- protected boolean canHaveEmptyValues = true

Constructors:

- public ContextView(AttrTupleManager manager, int mapStyle) Creates a new root context and returns a full access view for it.
- public ContextView(AttrTupleManager manager, int mapStyle, AttrContext parent) Creates a new child context and returns a full access view for it.
- public ContextView(AttrTupleManager manager, AttrContext source, boolean leftRuleSide) Returns a new view which shares another view's context and has the specified access mode;
- public ContextView(AttrTupleManager manager, ContextCore source) Returns a new view for the specified context.

Implementing SymbolTable:

- public HandlerType getType(String name) Implementing the SymbolTable interface for type retrieval.
- public HandlerExpr getExpr(String name) Implementing the SymbolTable interface for value retrieval.

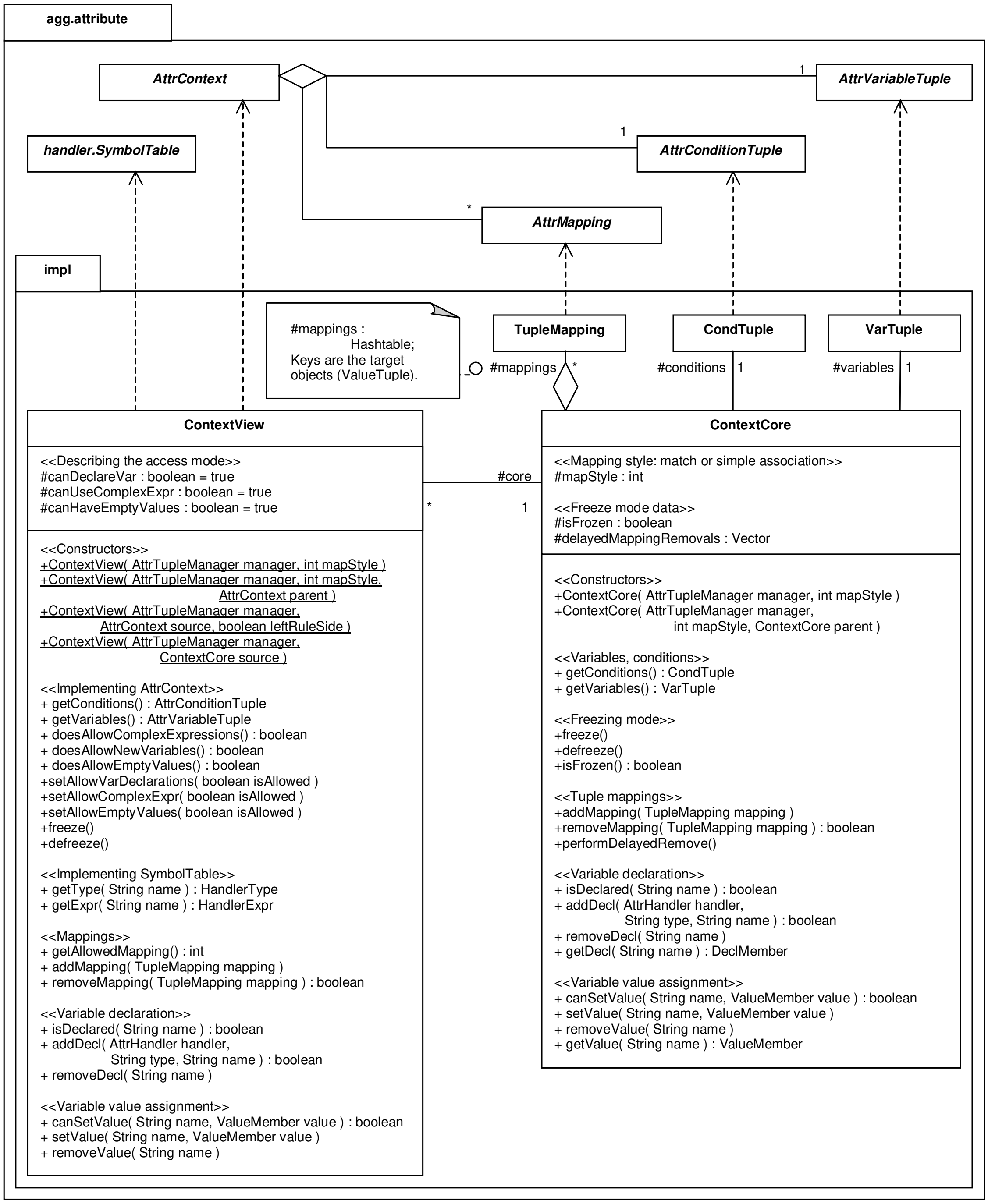

Figure 7.8: The attribute context implementation

The 'doesAllow...' methods test if the respective flags are set. The 'setAllow...' methods set these flags.

All the other methods are implementing the AttrContext interface or providing services for the attribute package by delegating to methods in ContextCore with the same names. These methods are described there.

ContextCore Contains declarations and values of context variables; Note that this class does NOT implement the AttrContext interface; The actual implementation for it is the wrapper class ContextView, which allows to restrict access to a context while sharing the actual state (variables, conditions, mappings). An example are the different access rights of the left and the right rule side in graph transformation.

Variables:

- protected int mapStyle Kind of mapping in this context, PLAIN_MAP or MATCH_MAP.
- transient protected boolean isFrozen = false Flag; when set, mapping removals are deferred.
- transient protected Vector delayedMappingRemovals Container with deferred mapping removals.

Constructors:

- public ContextCore(AttrTupleManager manager, int mapStyle) Creating a new root context.
- public ContextCore(AttrTupleManager manager, int mapStyle, ContextCore parent) Creating a new child context.

Variables, conditions:

- public CondTuple getConditions() Returns the condition tuple.
- public VarTuple getVariables() Returns the variable tuple.

Freezing mode:

- public void freeze() Switching on of the freeze mode; mapping removals are deferred until 'defreeze()' is called.
- public void defreeze() Perform mapping removals which were delayed during the freeze mode.
- public boolean isFrozen() Checking if this context is "frozen".

Tuple mappings:

- public void addMapping(TupleMapping mapping) Adding a new mapping to this context.
- public boolean removeMapping(TupleMapping mapping) Removing a mapping. Returns 'true' if the mapping was contained in this context at all, 'false' otherwise.
- private void performDelayedRemove() Removing the mappings in the 'delayedMappingRemovals' container.

Variable declaration:

- public boolean isDeclared(String name) Tests if a variable has already been declared in this context or in any of its parents.
- public boolean addDecl(AttrHandler handler, String type, String name) Adding a new declaration; "name" is a key and must not have been previously used for a declaration in this context or any of its parents. Returns 'true' on success, 'false' otherwise.
- public void removeDecl(String name) Removing a declaration from this context; Parent contexts are NOT considered; Does nothing if the variable "name" is not declared.
- public DeclMember getDecl(String name) Getting the declaration (type) of a variable.

Variable value assignment:

- public boolean canSetValue(String name, ValueMember value) Checking if a variable can be set to a given value without violating the application conditions. Note: if the conditions are violated already, this method returns 'true' for any 'value', unless 'value' contradicts a previously set non-null value for the variable.
- public void setValue(String name, ValueMember value) Setting a variable value.
- public void removeValue(String name) Removing a variable.
- public ValueMember getValue(String name) Getting the value of a variable.

TupleMapping Representation of a mapping between two attribute instances. The details are given in the source file comments and are not essential in order to understand the context implementation mechanism in principle.

7.3 Tuple Observer Mechanism

This section deals with the implementation of the observer mechanism. Figure 7.1 on page 113 gives an impression of the requirements involved. An attribute tuple can be an observer as well as an observable (or subject). This quality is obtained through the class ChainedObserver which is central to the observer mechanism. All attribute tuple classes are subclasses of ChainedObserver.

Figures 7.9 and 7.10 show the implementation of the observer mechanism. The first of the two illustrates the context of the class ChainedObserver and indicates the role of attribute tuple events. The second figure shows how the properties of ChainedObserver are extended and used by the tuple classes. The explanations of the participating classes follow below.

AttrEvent, AttrTuple, AttrObserver These interfaces are described in section 5.6 on page 75ff.

AttrObject Provides some convenience operations for its subclasses. Very useful for debugging. For more about debugging output see also the classes AttrSession and VerboseControl in this package.

ManagedObject This intermediate class was designed without a special purpose. It was thought, at one point all the attributes might want to access a central institution. So far, there was no need for this.

TupleEvent Implementation of AttrEvent. Whenever change events about a range of indices occur, the index range is described as [index0 .. index1].

ChainedObserver The central class of the observer mechanism. It is extended by all tuple classes, so their instances can be observers as well as subjects.

Preventing endless loop events:

- protected static final int MAX_SIZE_OF_EVENT_STACK = 100
- protected static int sizeOfEventStack = 0

Instance variables:

- protected transient Vector observers = new Vector(10, 10) Container with observers of this instance, all of which implement the AttrObserver interface.

Handling of observers (part of the implementation of *AttrTuple*):

- public Enumeration getObservers() Returns an enumeration of this instance's observers.

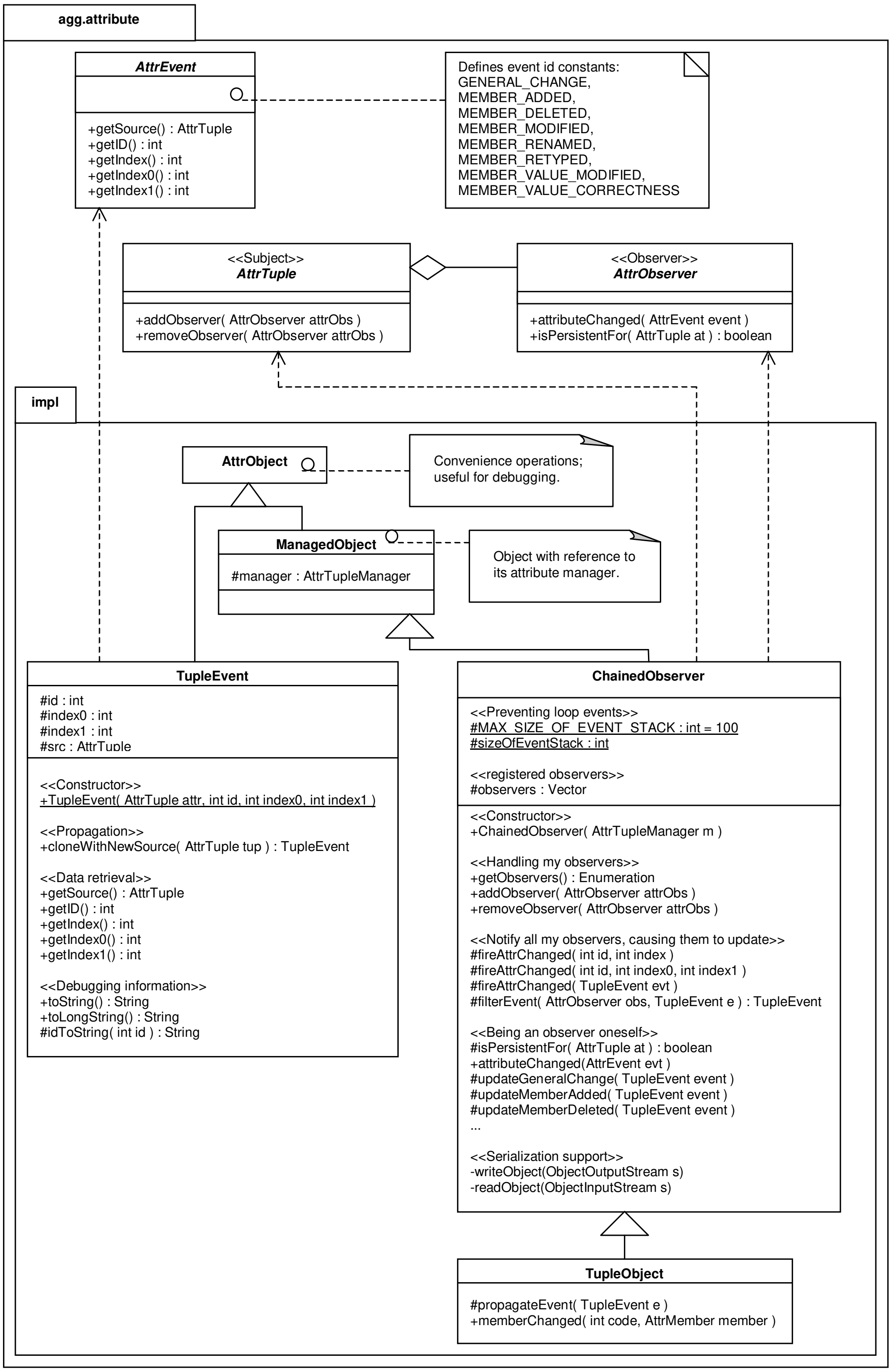

Figure 7.9: The class hierarchy of the observer mechanism

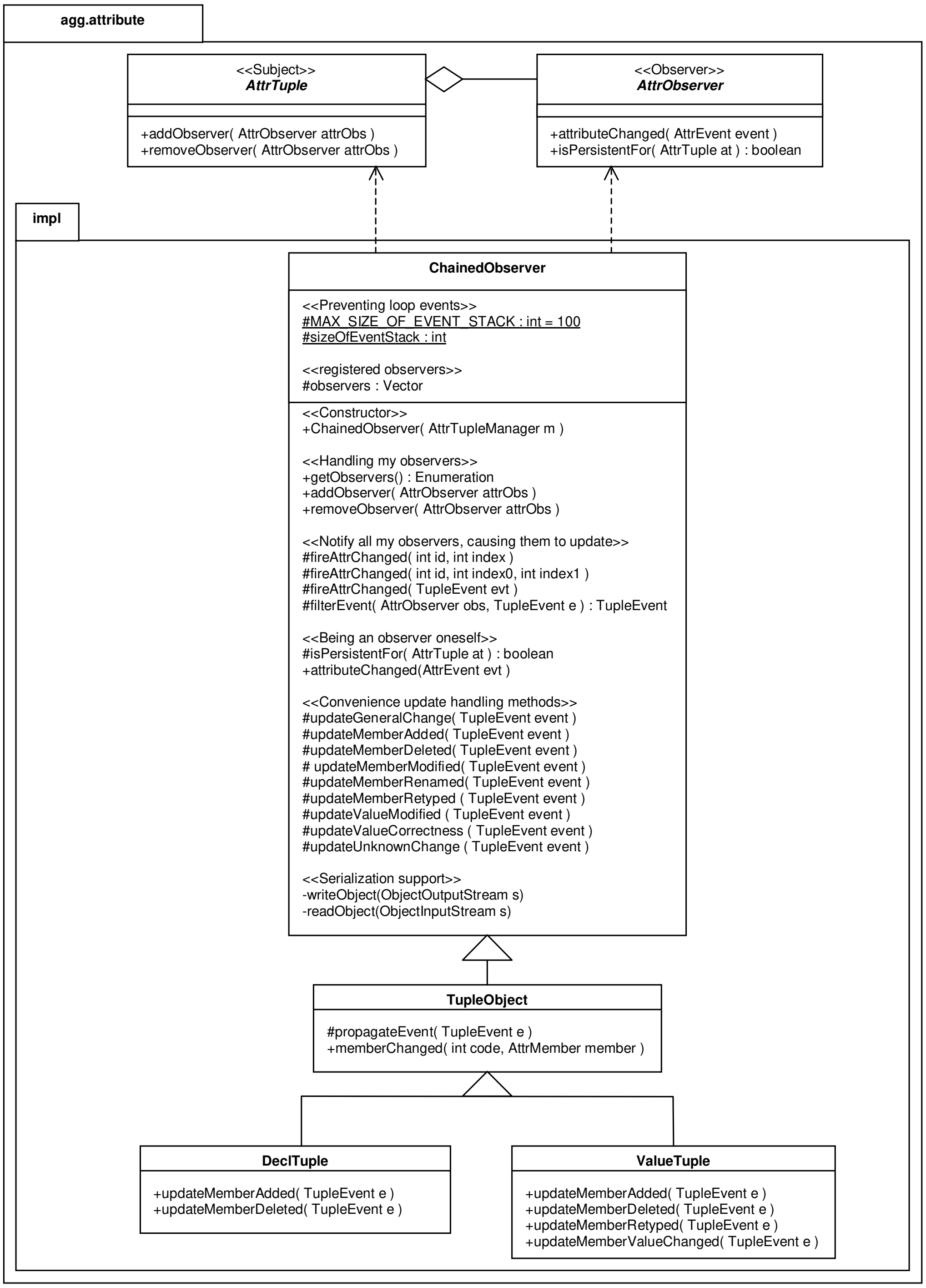

Figure 7.10: Tuples as observables within the class hierarchy

- public void addObserver(AttrObserver attrObs) Adds a new observer to the observer container.
- public void removeObserver(AttrObserver attrObs) Removes an observer from the observer container.

Notify all my observers, causing them to update:

- protected void fireAttrChanged(int id, int index) Convenience method for firing an event with the event id 'id' and the range [index .. index].
- protected void fireAttrChanged(int id, int index0, int index1) Convenience method for firing an event with the event id 'id' and the range [index0 .. index1].
- protected void fireAttrChanged(TupleEvent evt) Fires the event 'evt', which means that the method 'attributeChanged()' is invoked in every observer of this instance. The event is previously 'filtered' using the internal 'filterEvent()' method.
- protected abstract void propagateEvent(TupleEvent e) Propagates the event to the observers, pretending to be the source.
- protected TupleEvent filterEvent(AttrObserver obs, TupleEvent e) This method is called internally before notification of an observer. It performs the appropriate event transformation for the respective observer. The default implementation returns the same event unchanged. This method can be overridden by classes that wish to customize or filter the actual event depending on the respective observer and/or its own framework (index transformation, id change). If [null] is returned, the specified observer will not get any notification this time.

Being an observer oneself:

- public boolean isPersistentFor(AttrTuple at) Per default, always save observer dependencies within the attribute component.
- public void attributeChanged(AttrEvent evt) Checks if an endless event recursion took place. If so, a runtime exception with a warning text is thrown, as this indicates an error in the implementation. Otherwise, it calls the corresponding updating method that can be overridden by subclasses.

Convenience update handling methods:

- updateGeneralChange(TupleEvent event) - updateUnknownChange(TupleEvent event):

  The default implementations fire the same event, i.e. they simply pass it on to the observers of this instance. Subclasses simply override those of these methods where they want to implement their customized behaviour. In this manner, they do not need to write a dispatching loop, the dispatching mechanism is already implemented in the attributeChanged(AttrEvent evt) method.

Serialization support:

- private void writeObject(ObjectOutputStream s) Apart from calling the standard serialization procedure, saves the attribute observers which so desire (Those who return 'true' for `isPersistentFor( this )`).
- private void readObject(ObjectInputStream s) Apart from calling the standard serialization procedure, restores the attribute observers.

TupleObject Described in section 7.1.3 on page 116ff. Here, its contribution to the observer chain mechanism is presented:

- protected void propagateEvent(TupleEvent e) Implements the abstract method of 'ChainedObserver'. Propagates the event to the observers, pretending to be the source.
- public void memberChanged(int code, AttrMember member) Handles a member-event by finding the index of the given 'member' parameter in this tuple and calling fireAttrChanged(). This method is called by the members of this tuple after a change.

DeclTuple Described in section 7.1.4 on page 120ff. Here, its contribution to the observer chain mechanism is presented:

- public void updateMemberAdded(TupleEvent e),
- public void updateMemberDeleted(TupleEvent e):

  Reacts to events from the parent or an interface, propagating to my own observers (children, value tuples) and pretending to be the source.

ValueTuple
- public void updateMemberAdded(TupleEvent e) Event handling when a member was added. If the event comes from its declaration tuple, this method adds a new member. Propagates the event in any case.
- public void updateMemberDeleted(TupleEvent e) Event handling when a member was deleted. If the event comes from its declaration tuple, deletes the corresponding member and propagates the event.
- public void updateMemberRetyped(TupleEvent e) Event handling when a member was retyped. If the event comes from its declaration tuple, notifies the corresponding member of the type change. Propagates the event in any case.
- public void updateMemberValueChanged(TupleEvent e) Occurs when distribution is involved. Copies the value of the relevant member and propagates the event.

7.4 Views

The figure 7.11 illustrates the requirements of an attribute tuple view. It takes the object diagram of figure 7.1 on page 113 and extends it to include the views ':OpenViewSetting' and ':MaskedViewSetting' that observe the value tuple. These views are brokers for editors that wish to monitor this value tuple in various ways. Thus, 'fullEditor' has access to all the value members, while 'smallEditor' hides the member 'date of birth'.

The design of the view mechanism is described in section 5.3 on page 59ff. What follows, is the presentation of how it is implemented.

Figure 7.11 also shows how an instance of 'OpenViewSetting' has a table of format descriptions, where for every declaration tuple there are two lists:

- allSlots describes the permutation of the tuple members, where all members are included.
- visibleSlots describes the same permutation, but hidden members are left out.

Thus, 'OpenViewSetting' is the actual administrator of the tuple layout, while 'MaskedViewSetting' has access to it via the 'visibleSlots' - container. Both view settings, however, have their individual lists of tuple observers, in this case the presented editors.

Figure 7.12 shows the class hierarchy of the view implementation, specifically how the responsibilities are shared between the classes ViewSetting, OpenViewSetting and MaskedViewSetting.

Figure 7.13 shows in more detail how the permutation and hiding of tuple members is implemented using the classes TupleFormat, SlotSequence and SlotSequence.Slot.

The explanations of the participating classes follow below.

AttrViewSetting, AttrObserver These interfaces are described in section 5.6 on page 75ff.

ViewSetting Common superclass for OpenViewSetting and MaskedViewSetting. Provides most routines for handling own observers and event propagation. Most methods that actually manipulate the layout of attribute tuples are in its subclasses.

OpenViewSetting This is the central class in the view mechanism implementation. It is observing the attribute tuples, implementing permutation and hiding of tuple members using a table of tuple formats, one for each tuple declaration whose value tuples it observes. When a change notification occurs, the indices are converted and a corresponding 'TupleViewEvent' is 'sent' to the affected value tuple's observers and the 'MaskedViewSetting' is notified as well. The slots are referring to the layout where all tuple members are visible.

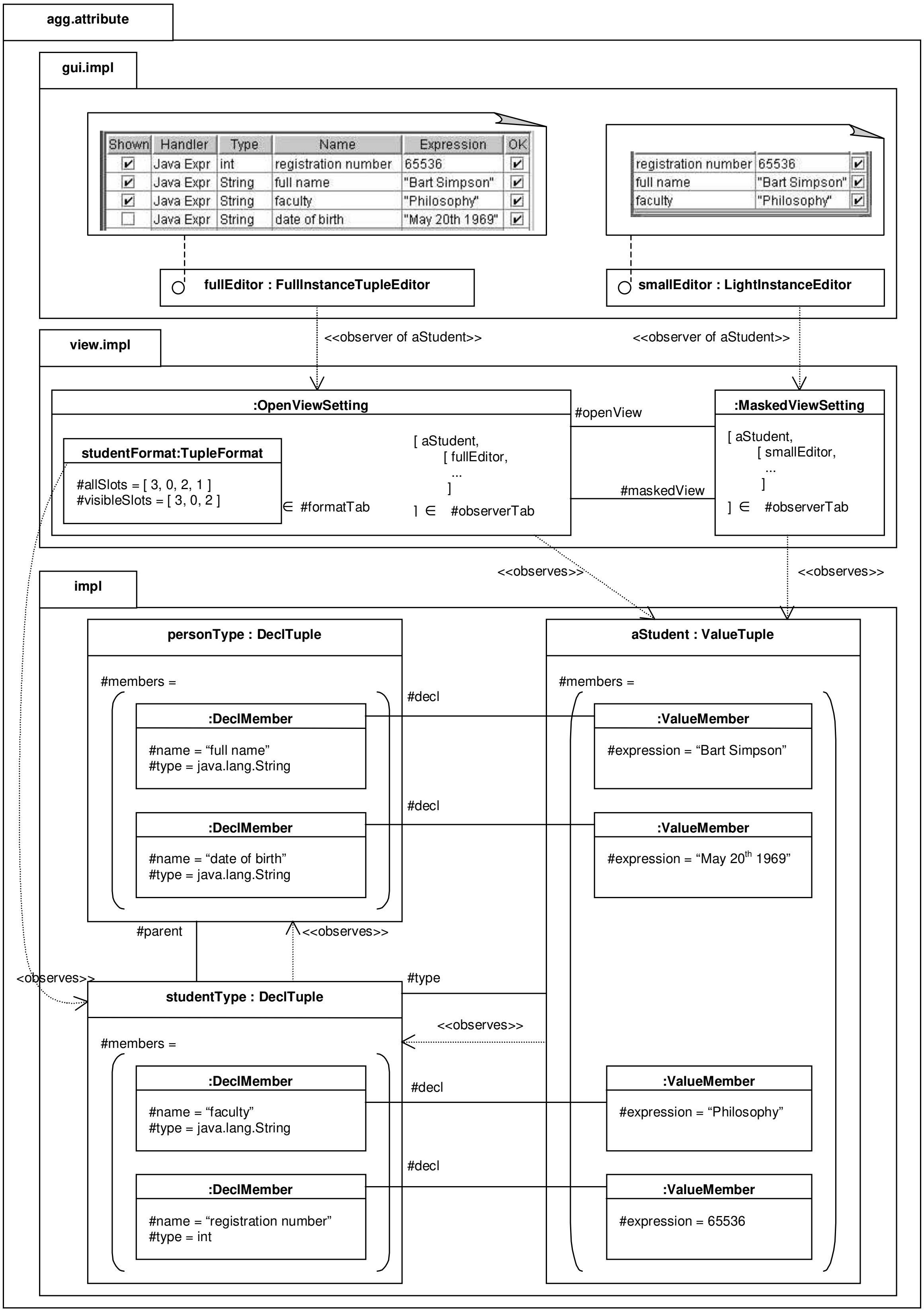

Figure 7.11: Example instance diagram with views as observers

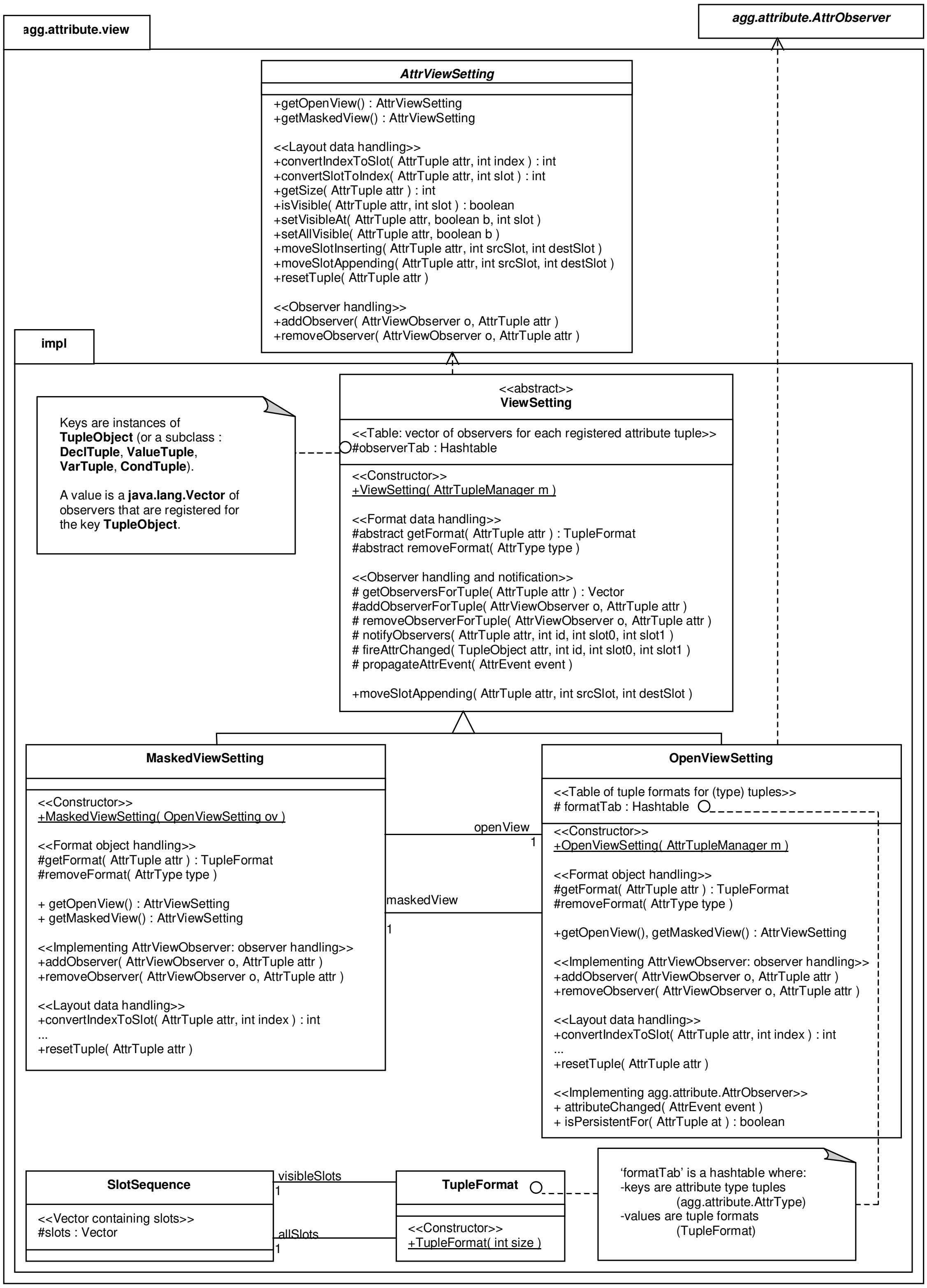

Figure 7.12: Class hierarchy of the view setting implementation

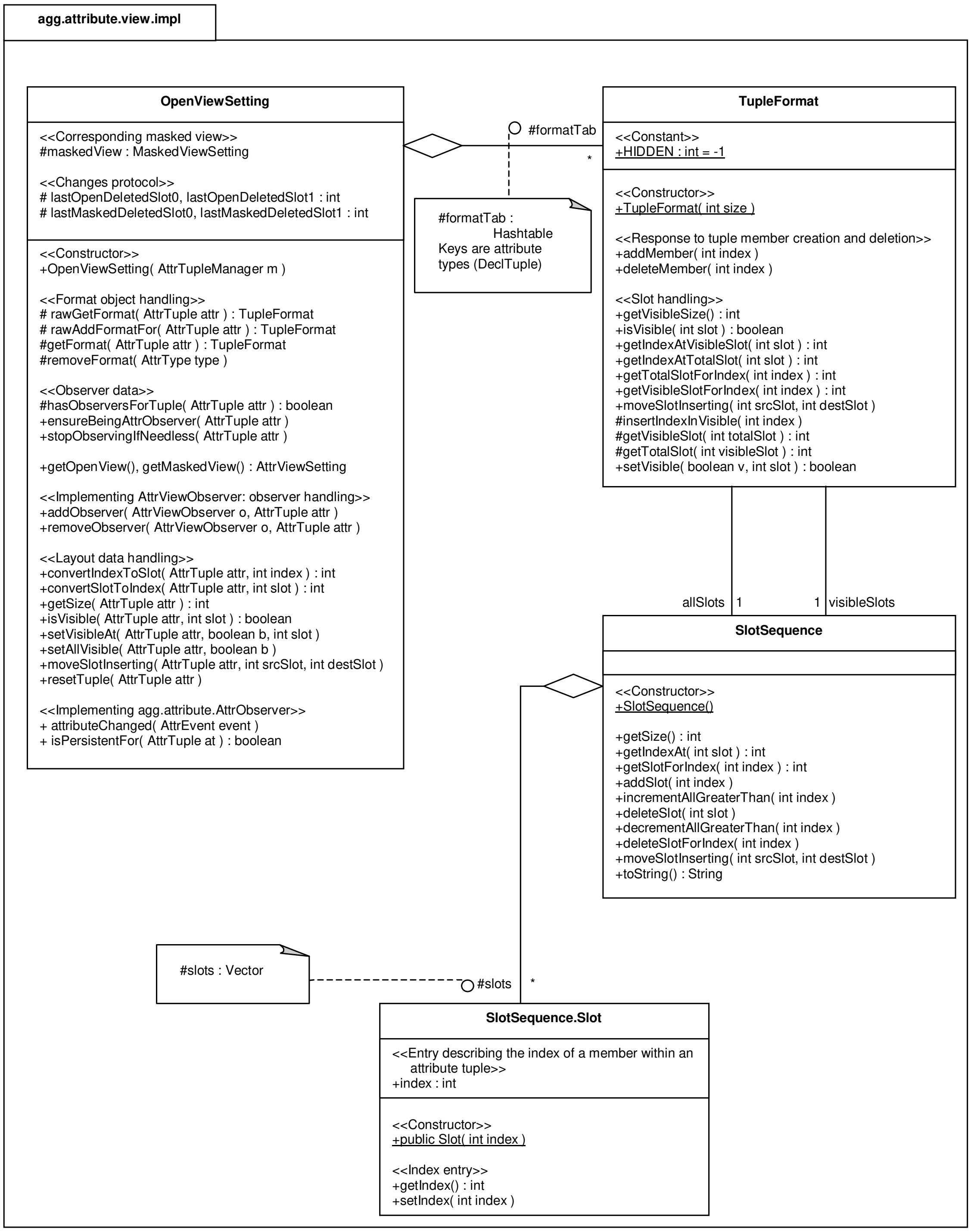

Figure 7.13: View setting implementation

MaskedViewSetting This class delegates the execution of most of its methods to its OpenViewSetting instance. The slots are referring to the layout where only non-hidden tuple members are visible, and so the slot/index conversion implementation is different from that of OpenViewSetting.

TupleFormat Format of an attribute tuple. Within instances of OpenViewSetting, each declaration tuple has a TupleFormat instance assigned to it. Instances of TupleFormat each have two instances of SlotSequence, one for all the tuple members and one for just the visible ones. The main task of TupleFormat is to encapsulate and synchronize the handling of these two 'parallel' structures, to keep them consistent and to provide service methods for OpenViewSetting and MaskedViewSetting for adding, inserting, deleting of slots and for converting between member indices and slots.

SlotSequence This class implements a mapping between member indices and tuple layout slots. As for now, it uses a vector of Slot instances. Each Slot instance contains a member index. That means, the position of an index in this vector is its layout slot.

SlotSequence.Slot Instances of this inner class are used as container elements for SlotSequence.

7.5 Editors

This section describes how the graphical user interface design in section 5.9 on page 85ff is implemented by a hierarchy of flexible editors.

Figure 7.14 shows how the GUI design is paralleled and contrasted by the editor implementation structure. Obviously, the editor interface inheritance structure for *AttrTopEditor* that combines a tuple editor, a context editor and a customizing editor is implemented using aggregation, with TopEditor implementing the *AttrTopEditor* interface. Moreover, the tuple editor has a deep inheritance hierarchy. By offering a new piece of reusable features in each layer, this approach enables the developer to realize new editors very quickly. And these are the classes of the diagram:

AttrEditor, AttrCustomizingEditor, AttrContextEditor

AttrTupleEditor, AttrTopEditor, AttrEditorManager These interfaces are described in section 5.9 on page 85ff.

EditorManager Implementation of the abstract factory interface 'AttrEditorManager'. This class has just one single instance (Singleton Pattern) that can be obtained by invoking 'EditorManager.self()'. From that instance, one can get instances of the editors as specified in the 'AttrEditorManager' interface.

AbstractEditor Abstract root class for all editors.

TopEditor Combination of a context editor, a full instance editor and a customizing editor of an attribute manager. The method implementations are delegating, most of the time, to the respective sub-editors.

CustomizingEditor Customizing of an attribute manager as well as his handlers. The implementation uses a tabbed view.

ManagerCustomizingEditor Customizing of an attribute manager.

ContextEditor Editor for a context (rule, match etc.). It consists of a variable tuple editor and a context tuple editor.

BasicTupleEditor Provides all necessary functionality for a lightweight editor of an attribute tuple. Extending classes just need to redefine createTableModel() to set up a simple editor with desired columns, headers etc. For row dragging, tool bar actions etc. consider extending 'ExtendedTupleEditorSupport'.

LightInstanceEditor Editor for selected data of an attribute instance tuple. However, it does not complain if a type tuple is edited instead. No possibility of adding/deleting of members.

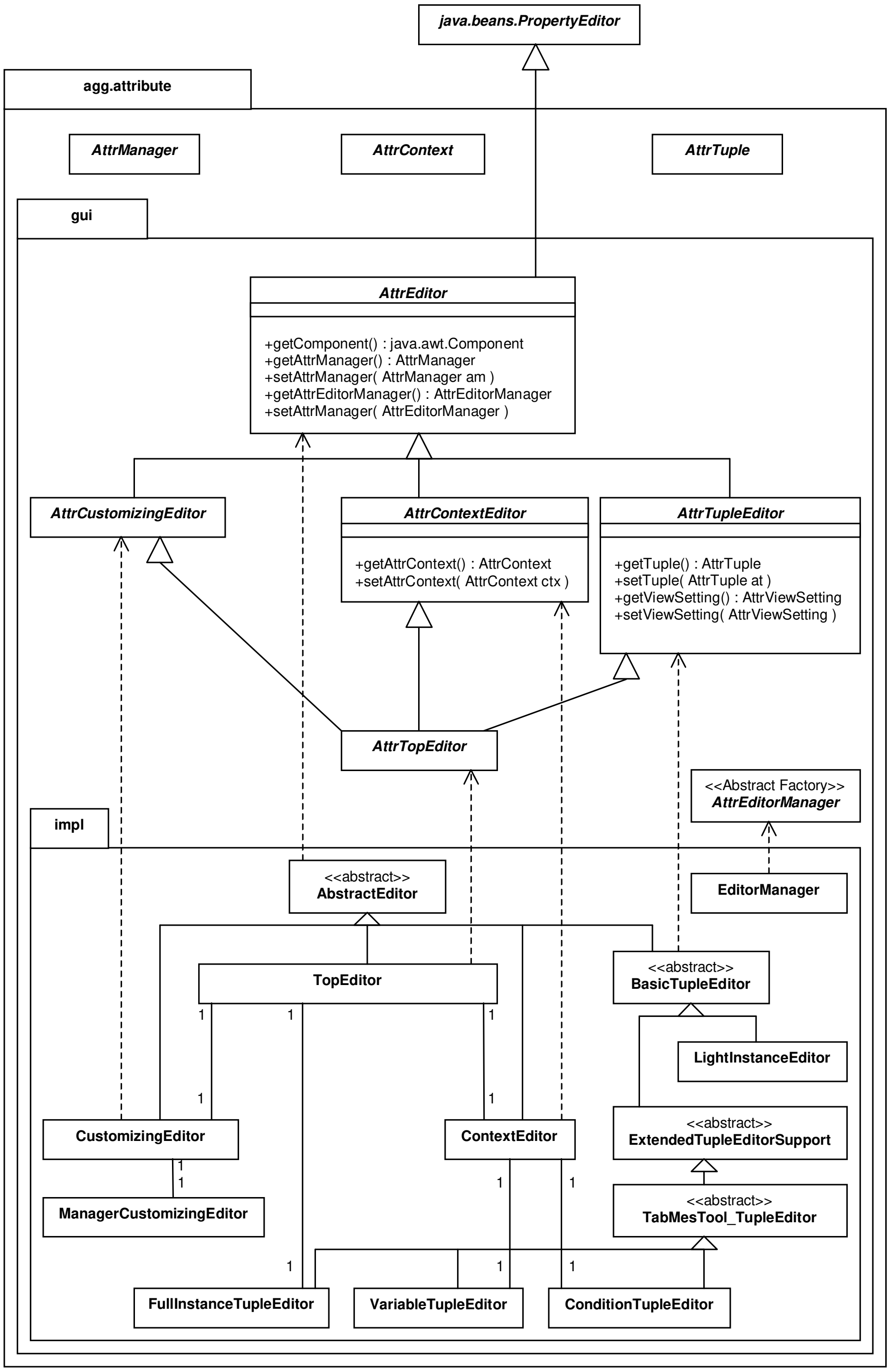

Figure 7.14: Editor implementation architecture

ExtendedTupleEditorSupport Abstract editor for all data of a tuple. Implements various 'actions' like dragging (changing the order) of rows (i.e. members) in a tuple, deleting, hiding or evaluating of members etc. and a message area. Extending classes can select from features by simply adding the respective actions (instances of class 'Action', obtained by calling, e.g. 'getDeleteAction()', to a tool bar. Row dragging can be turned on or off using 'setRowDraggingEnabled(boolean b)'. The message area with error and warning messages can be created invoking 'createOutputTextArea()'. Examples of how to 'construct' a customized editor in the described way can be found in the extending classes: 'TabMesTool_TupleEditor', 'FullInstanceTupleEditor', 'VariableTupleEditor', 'ConditionTupleEditor'.

TabMesTool_TupleEditor Abstract editor, providing the following general layout. From top to bottom:

1. The table view.
2. Message text area.
3. Button panel.

The set of features can be 'constructed' by overriding the generic method 'createToolBar()'.

FullInstanceTupleEditor Editor for all data of an attribute instance tuple.

VariableTupleEditor Editor for a variable-tuple.

ConditionTupleEditor Editor for a condition tuple.

Figure 7.15 presents the abstract level of the tuple editor hierarchy.

Figure 7.16 shows how the class TupleTableModel is employed in order to customize the table of a tuple editor in a flexible way. A table is used in order to render and edit an attribute tuple and its members.

JTable A Swing class, realizing a table with a number of rows and columns, where table values can be made editable. The layout and other properties of such a table is largely dependent on the configuration of the underlying *TableModel* instance.

TableModel The Swing interface defining the underlying model for a JTable.

AbstractTableModel A Swing class that helps implementing the *TableModel* interface.

TupleTableModel Table model for tuple editors. The following behavior can be customized by method calls: which columns to display, their order, their titles, classes of their content objects and if their objects are editable.

TupleTableModelConstants Constants for accessing properties of a tuple table model. So far, all of them are column IDs.

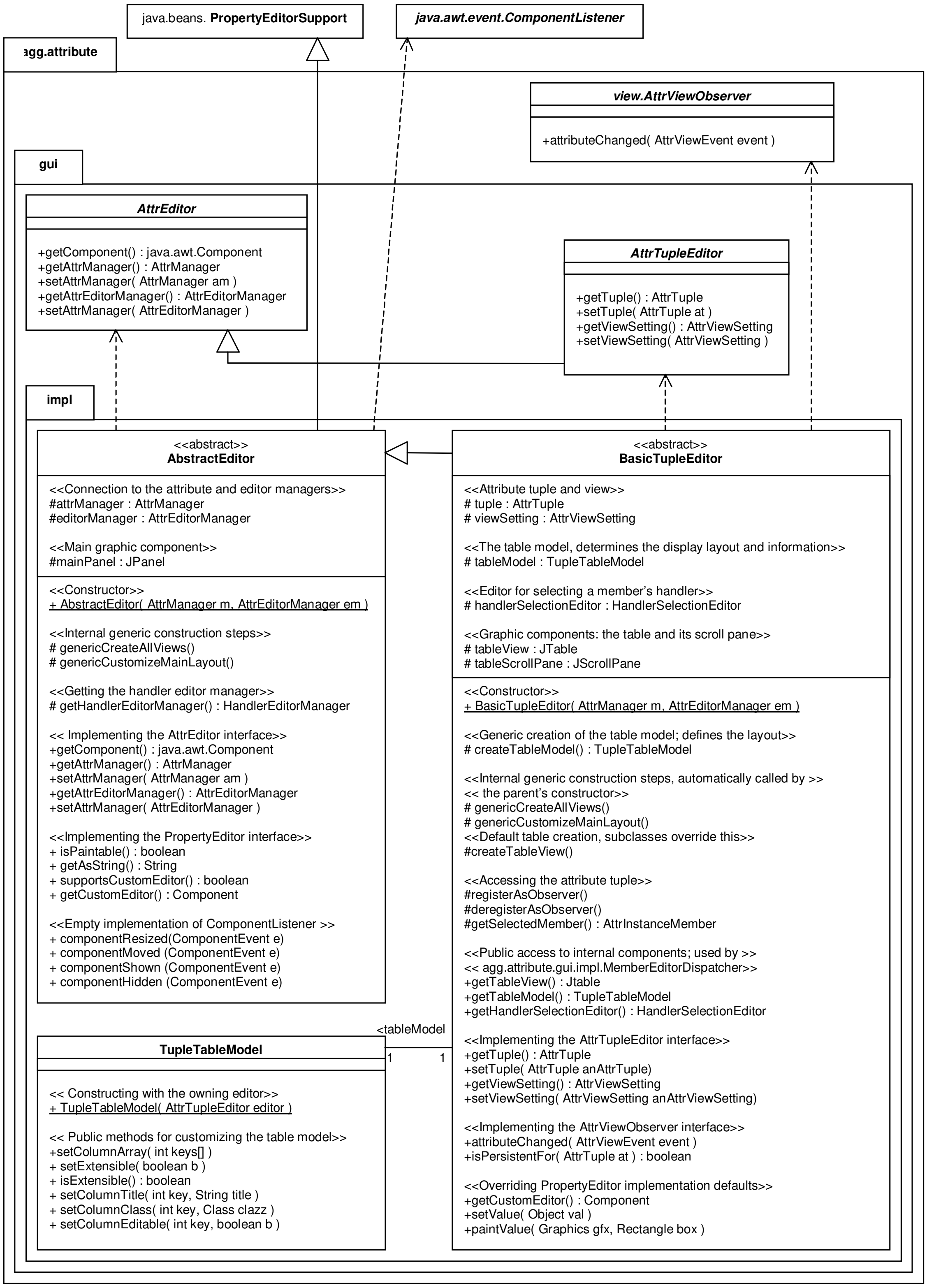

Figure 7.15: Basic tuple editor class structure

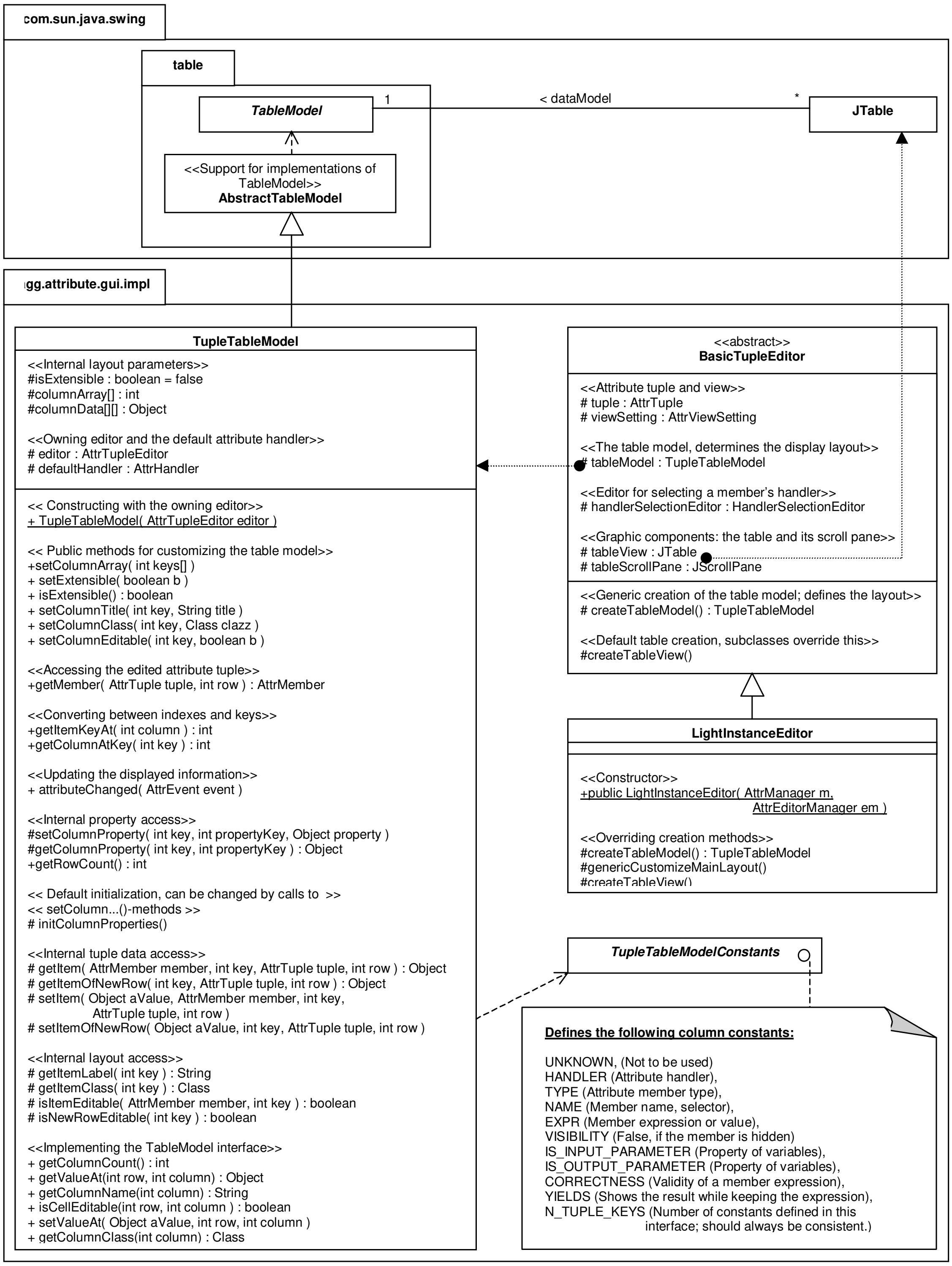

Figure 7.16: Tuple table model

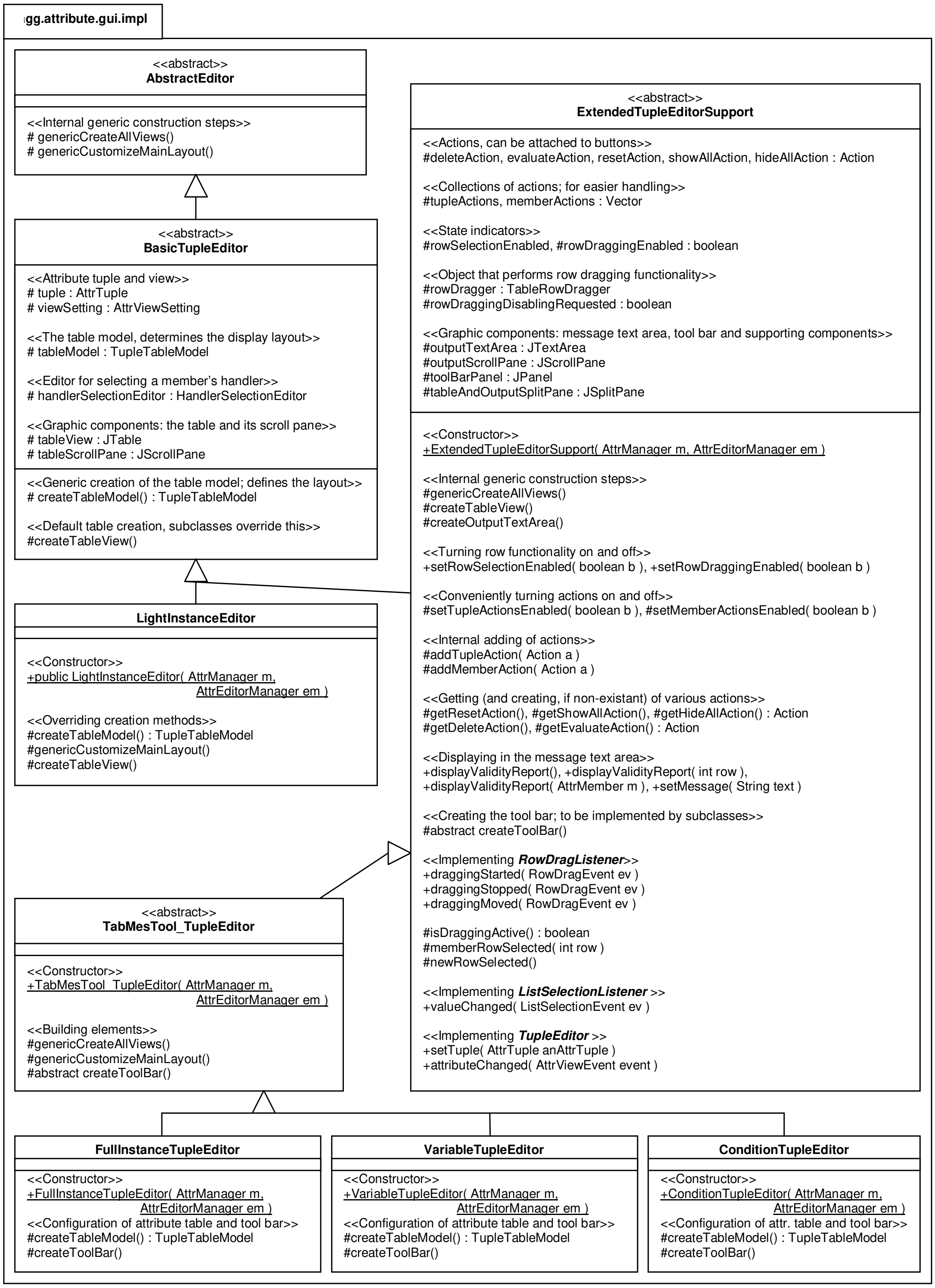

Figure 7.17: Tuple editor inheritance hierarchy

Figure 7.17 shows how other features of a tuple editor are added on. These features include for example a message area, buttons for deleting, hiding or evaluating of members etc. One can see how little the actual concrete editors (FullTupleEditor, VaribleEditor and ConditionEditor) need to do in order to obtain their respective features. They merely pick and choose from the 'pool' of properties that are available to them through inheritance.

7.6 Java Handler and its Interpreter

The package agg.attribute.handler.impl.javaExpr contains classes whose names indicate which interfaces of the package agg.attribute.handler they implement:

Interface in `agg.attribute.handler`	Implementing Class in `impl.javaExpr`
AttrHandler	JexHandler
HandlerType	JexType
HandlerExpr	JexExpr

The implementations usually delegate the execution of tasks to methods in the class agg.attribute.parser.javaExpr.Jex.

The actual parser in agg.attribute.parser.javaExpr contains a number of classes. Their instances are nodes of the abstract syntax tree that is generated by the parser.[1]

The inheritance hierarchy is represented by way of indentation in the list below. Only the classes with the prefix *AST* are fully defined (not abstract) and their instances can be nodes of an abstract syntax tree. The methods that are shown in parentheses after a class are defined in that class. The `checkContext` method is equal for many arithmetic operations. It is therefore defined in the abstract classes.

[1]See the description of JavaCC and JJTree in section 3.4.

- Interface Node
  - SimpleNode (implements Node)
    - ASTAddNode (checkContext, interpret)
    - ASTAllocationExpression (checkContext, interpret)
    - ASTArrayAllocation
    - ASTArrayIndex
    - ASTCharConstNode (checkContext, interpret)
    - ASTClassName (checkContext, interpret)
    - ASTCondExpr (checkContext, interpret)
    - ASTEmptyDimension (checkContext, interpret)
    - ASTExpression (checkContext, interpret)
    - ASTFloatConstNode (checkContext, interpret)
    - ASTId (checkContext, interpret)
    - ASTIntConstNode (checkContext, interpret)
    - ASTMemberName
    - ASTModNode (checkContext, interpret)
    - ASTNullLiteral (checkContext, interpret)
    - ASTPrimaryExpression (checkContext, interpret)
    - ASTStringConstNode (checkContext, interpret)
    - BOOLtoBOOLnode (checkContext)
      - ASTNotNode (interpret)
    - BOOLxBOOLtoBOOLnode (checkContext)
      - ASTAndNode (interpret)
      - ASTOrNode (interpret)
    - BoolNode (checkContext)
      - ASTFalseNode (interpret)
      - ASTTrueNode (interpret)
    - MemberNode
      - ASTField (checkContext, interpret)
      - OpMemberNode (checkContext, interpret)
        - ASTMethod
        - ASTAction (checkContext, interpret)
    - NUMtoNUMnode (checkContext)
      - ASTBitwiseComplNode (interpret)
      - ASTNegNode (interpret)
    - NUMxNUMtoBOOLnode (checkContext)
      - ASTGENode (interpret)
      - ASTGTNode (interpret)
      - ASTLENode (interpret)
      - ASTLTNode (interpret)
    - NUMxNUMtoNUMnode (checkContext)
      - ASTBitwiseAndNode (interpret)
      - ASTBitwiseOrNode (interpret)
      - ASTBitwiseXorNode (interpret)
      - ASTDivNode (interpret)
      - ASTMulNode (interpret)
      - ASTSubtractNode (interpret)
    - TYPE1xTYPE1toBOOL (checkContext)
      - ASTEQNode (interpret)
      - ASTNENode (interpret)

7.7 Testing Environment

This section briefly describes the testing environment and its implementation. The testing environment allows the developer to quickly try out, test or present something in the attribute component.

Figure 7.18 shows an annotated screenshot of the testing environment. It represents a simple implementation of a 'graph grammar' consisting of one rule, a set of graph types and a host graph. All graph nodes have an attribute tuple instance which is displayed using instances of LightInstanceEditor. In the lower right corner, there is the comprehensive umlCTopEditor, containing a full editor for the currently selected graph node, a context editor for the rule and a customizing editor for the attribute manager and the Java expression handler.

New types can be created using the 'New' button in the type window (top right). There are three graphs: the left rule side, the right rule side and the host graph. In order to create a new node in a graph, please select the desired type in the type window and then press the 'New' button in the graph window where the node should appear.

In order to create a copy of a node in a graph, please select the node to be copied and press the 'Copy' button in that graph window where the selected node should be copied into.

The 'Order' buttons reorder the nodes in a graph.

The usage of the editors is described in chapter 8.

Figure 7.19 shows the implementation structure of a primitive graph grammar (GraGra) with graphs (Graph), rules (Rule) and node types (GrObjType). There are also nodes (GrObjInstance) in graphs and a rule has a morphism (Morphism). It is clear that a GrObjType observes an *AttrType* and a GrObjInstance observes *AttrInstance*, while each morphism has an *AttrContext*.

Figure 7.20 shows how the graph grammar is being represented by GUI entities. Type graph editors (TypeGraphEd) contain type editors (GraTypeEd). Graph editors (GraphEd) contain node editors (GraInstEd). 'SwingTest' represents the whole graph grammar (GraGra). 'GraphEd' shows a graph (Graph). A type editor (GraTypeEd) represents a node type (GrObjType). A node editor shows a node (GrObjInst). 'SwingTest' uses a TopEditor in order to display and edit the rule context, the attribute system customization and the currently selected node. 'GraInstEd' uses LightInstanceEditor.

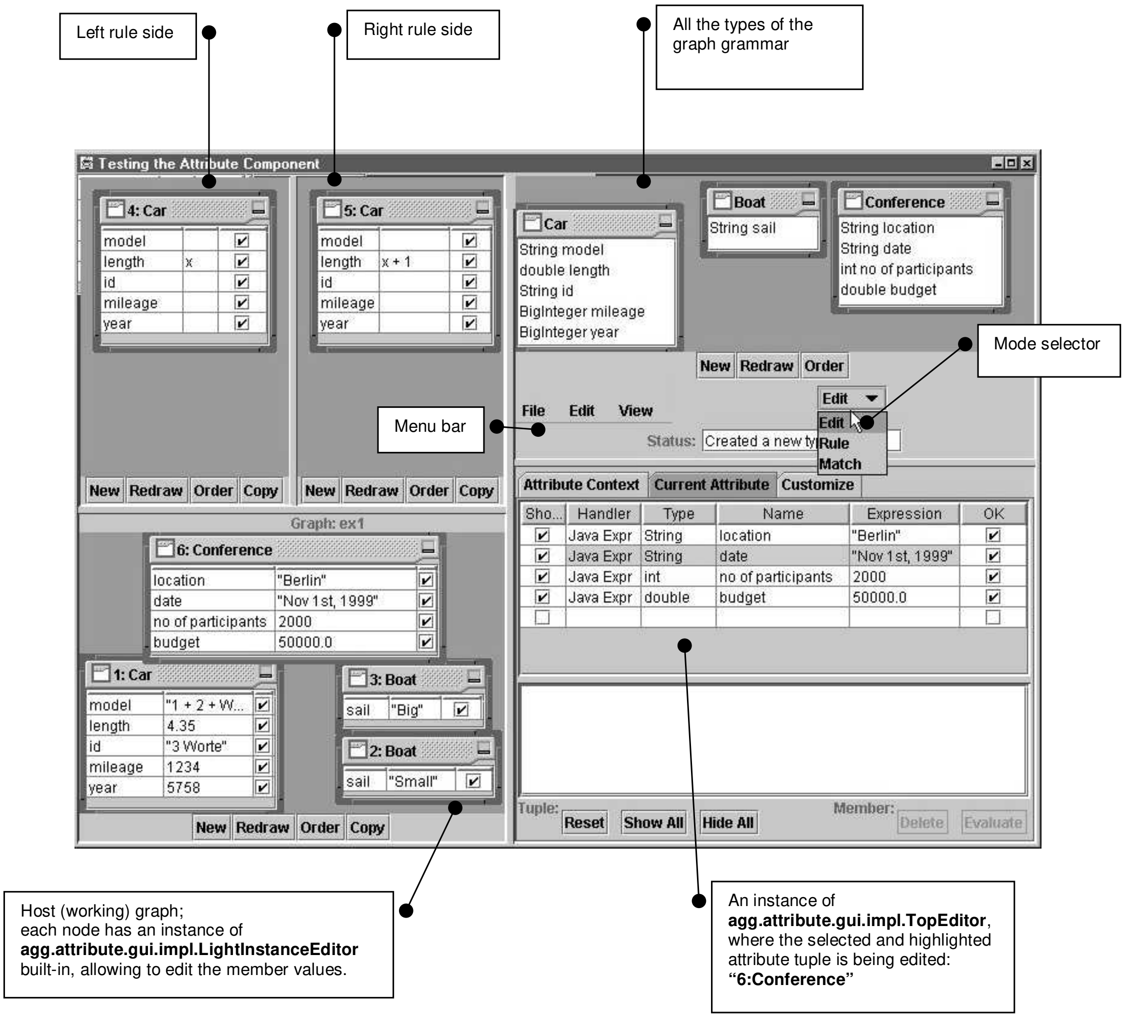

Figure 7.18: An annotated example screenshot of the testing environment

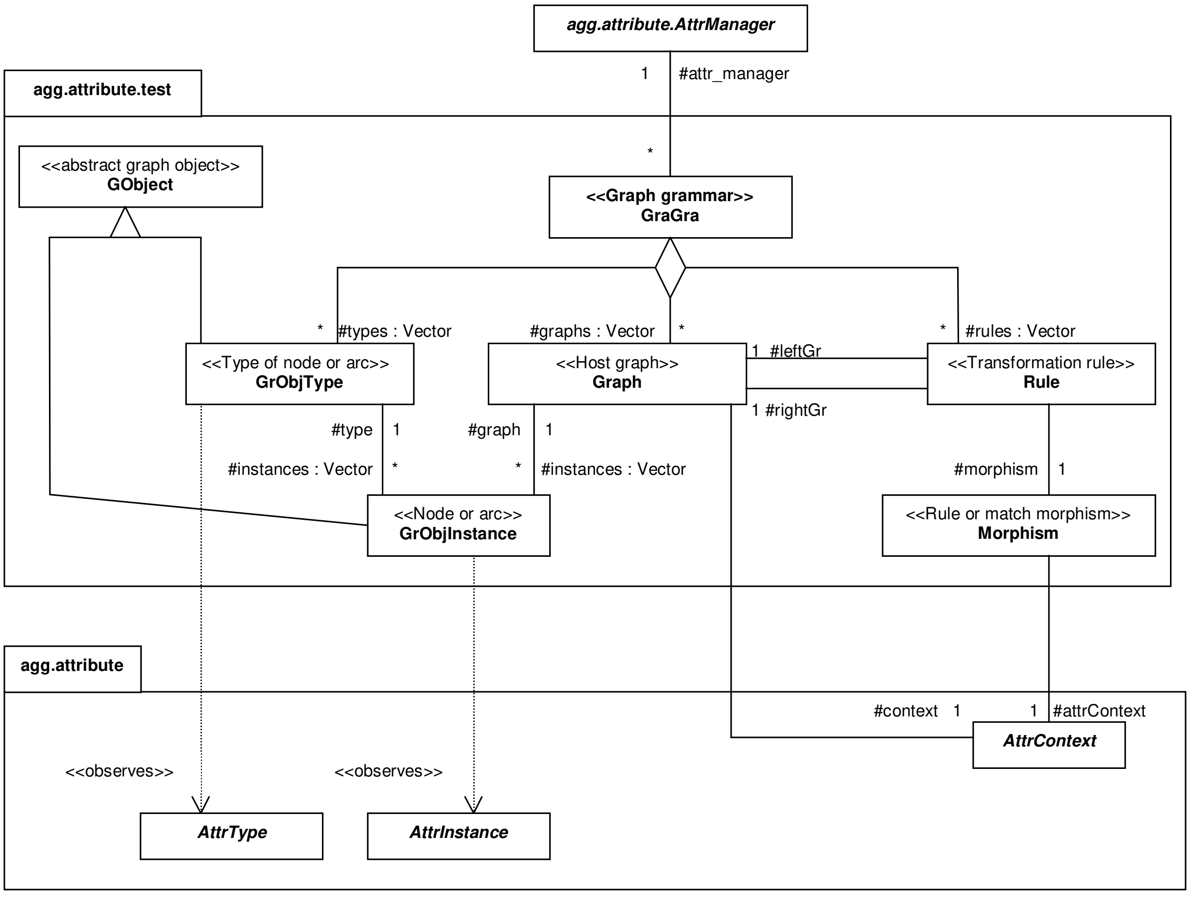

Figure 7.19: Graph grammar structures of the testing environment

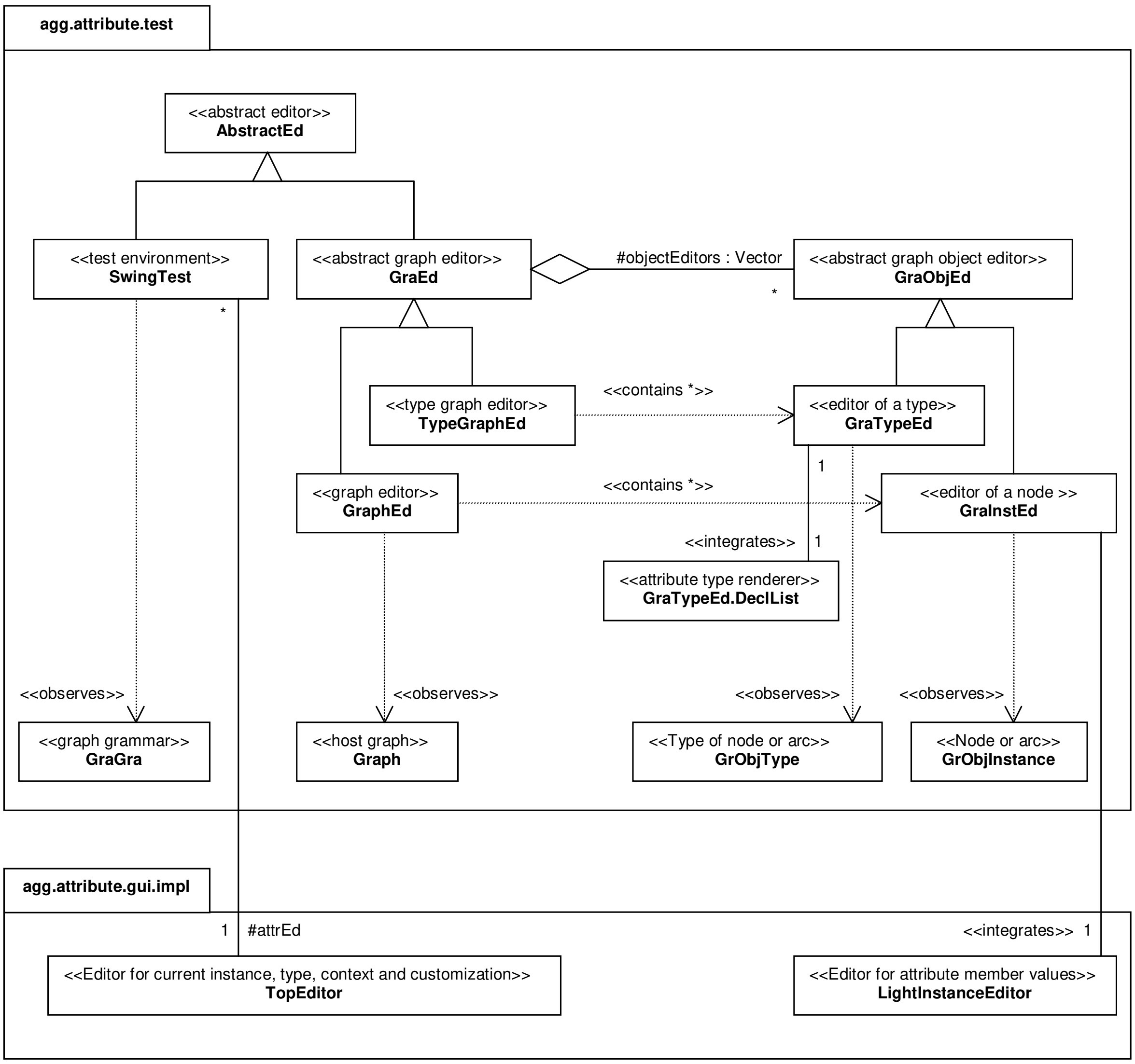

Figure 7.20: Editor structure of the testing environment

Chapter 8

Attribute Editor Manual

This chapter explains how to use the provided editors for attribute tuples, contexts and customization. Nearly all the editors are, graphically speaking, based on tables, containing lines and columns. Changing a cell's contents is done as follows:

- Clicking once on a cell can have one of the following effects:
  1. The cell gets a dark blue[1] background color. This means, the cell can be edited. If the user simply starts typing a text while the cell has the dark background, this text is appended to whatever contents the cell might already have.
  2. The cell gets a light blue background color, or its background remains uncolored at all. This is a sign, that this cell's contents cannot be changed using the editor.
- If a cell is editable, clicking twice marks its contents, and any text subsequently typed will replace the previous contents. The cursor can, however, be positioned freely within the cell, using the cursor keys or the mouse.

Additionally, the appearance of a table can be changed in the following ways:

Changing Column Positions By clicking on a column header and moving it while keeping the mouse button pressed, the order of columns can be changed.

Changing Column Width If the user wishes to change the with of a column, the narrow space between two column headers *right* of the considered column should be clicked and the mouse moved with the mouse button pressed down. If the mouse is moved to the right, the column becomes wider, while the required space is taken in equal parts from the other columns. If the mouse is moved to the left, the column becomes narrower, and the resulting additional space is added in equal parts to the other columns.

[1] The exact color tone may vary from system to system (blue, violet etc.); the real criterion is, if the background becomes dark colored or light colored.

8.1 Compact Instance Editor

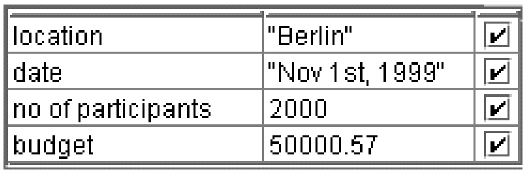

Figure 8.1: Compact instance editor

The compact instance editor which can be integrated within, e.g., a little box representing a graph node, is shown in figure 8.1. An example attribute tuple instance is "loaded" into the editor. The roles of the columns, from left to right, are as follows:

1. **Name**: The name (selector) of an attribute member within a tuple.
2. **Expression**: The expression of the instance member.
3. **OK**: Shows, if the member expression is a valid one, in the instance's context.

The values (or expressions) of the tuple can be edited, but not the names.

8.2 Comprehensive Editor

The comprehensive (or "Top"-) editor consists of several editors (context variables, context application conditions, the current attribute tuple etc.). They are partly using the same window space, which makes the editor system quite compact and suitable for integration into a client system. The sharing of space is organized using "filing cards". The editor is shown in figures 8.2 through 8.5. The three main parts, *Attribute Context*, *Current Attribute* and *Customize*, can be seen as card labels on top. Each part can be selected by clicking on its card label.

8.2.1 Tuple Editor

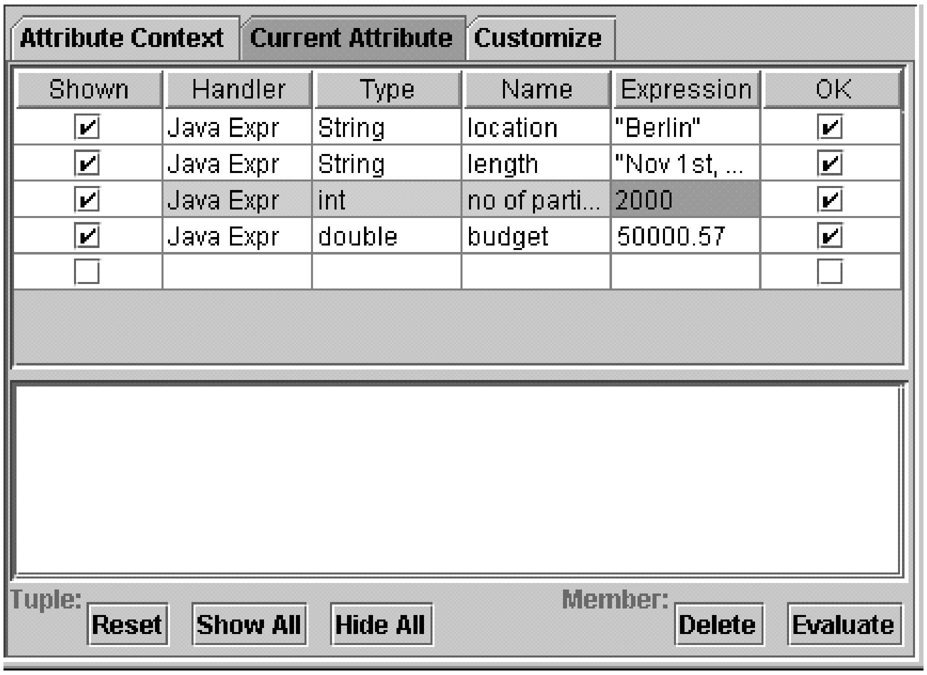

Figure 8.2: Attribute tuple editor

Figure 8.2 shows an attribute tuple being edited. The editor consists, from top to bottom, of:

1. a table, representing the tuple;

2. a text area, displaying help messages and error reports;

3. a button bar.

Description of the table: Each line but the lowermost represents a tuple member. The last line is empty and allows the user to add a new member by entering information in one of this line's editable cells. The columns, from left to right:

1. **Shown** indicates, if a member is shown in the compact editors (see previous section) or not. It can be turned on and off, and the changes will take effect immediately.
2. **Handler**: the attribute handler responsible for a member.
3. **Type**: the member's type, changeable;
4. **Name**: the member's name (selector), changeable;
5. **Expression**: the member's expression, changeable;
6. **OK**: Shows, if the member expression is a valid one, in the instance's context. It cannot be changed directly, since it is meant to react to changes in the member data.

Clicking (and thereby marking) a member line that does not have the OK-marker displays information about that member in the text area. If the member is valid, no information is displayed.

Lines can be moved up and down by clicking on a line and moving the mouse while holding the mouse button (dragging). By turning the "Shown" switch of a member on and off, this member can be hidden or shown. Since the mask layout is the same for every instance of a given type, the changes affect all instances of that type. The changes take effect immediately. While changing the order of lines (members) is visible in all editors, the tuple editor currently described never really hides the members. The hiding is done within the "compact editors" (previous section).

The buttons at the editor bottom:

Reset Resetting all layout changes (moving and hiding of members) to the initial state;

ShowAll Making all members visible at once;

HideAll Making all members invisible at once;

Delete Deleting the marked member (affects the tuple type and all its instances);

Evaluate Trying to evaluate the expression in the marked line, using the tuple's context.

8.2.2 Context Editor

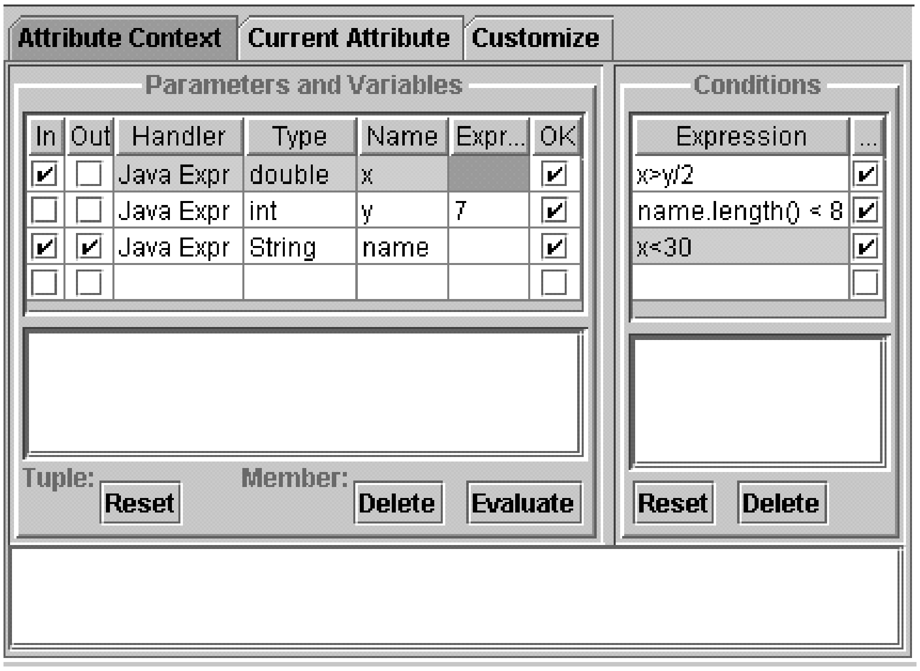

Figure 8.3: Context editor

Figure 8.3 shows the context editor. It consists of two tuple editors, one for the context's variables (left side) and one for its application conditions (right side). Below that, there is a text area for displaying specific messages.

Variables as well as conditions can be moved up and down in their tuples, but neither can be made invisible. The variable editor additionally has two columns prepended:

In is a switch for a variables parameter property; if it is checked, the variable in that line is an input parameter;

Out is a switch for a variables parameter property; if it is checked, the variable in that line is an output parameter;

The conditions have only two columns, the expression and the validity status, since the type is always *boolean* and names (selectors) are not needed.

The buttons at the bottom of these editors is a subset of those from the previous subsection.

8.2.3 Customizing Editor

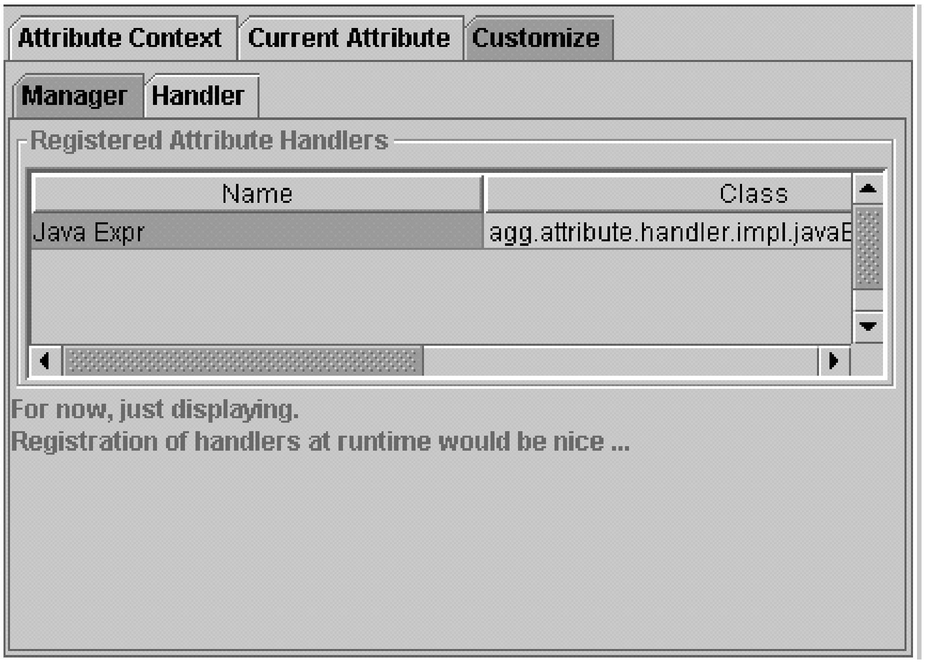

Figure 8.4: Customizing editor, manager part

Figures 8.4 and 8.5 show the customizing editor. It has two sub-editors: *Manager* and *Handler*.

The manager editor displays the registered attribute handlers. The columns are rather long and can only be seen using the horizontal scroll bars. They are:

Name The name to display in attribute tuple editors;

Class The class of the attribute handler;

Editor Class The class of the handler customizing editor;

The handler part momentarily contains just the Java expression handler. Additional handlers would appear with their card labels on the left side. The Java handler customization consists of defining which packages to search for classes. This can be compared to the set of imported packages in a Java program.

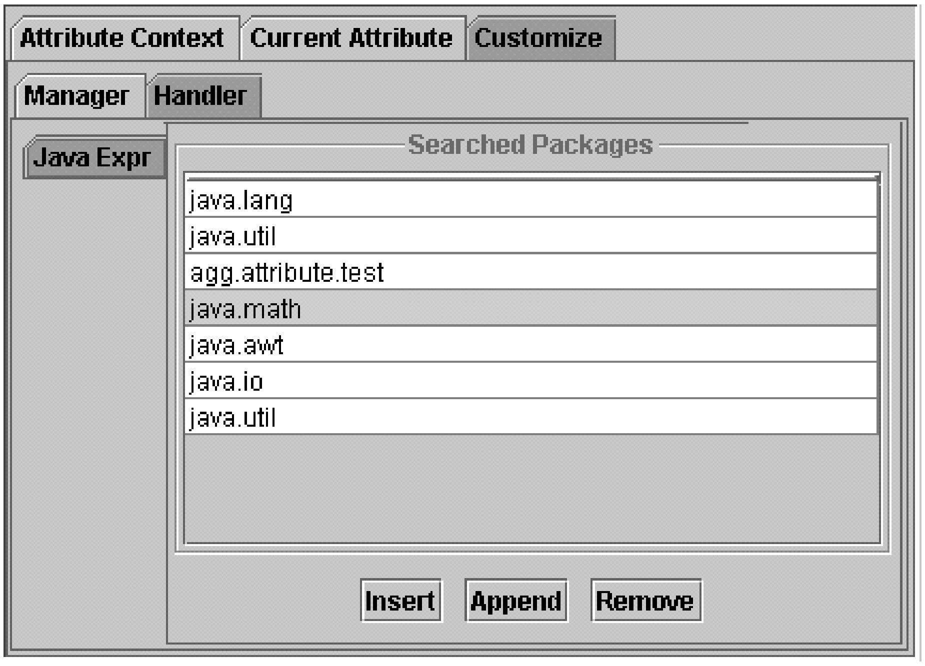

Figure 8.5: Customizing editor, handler part

Chapter 9

Conclusion and Outlook

This thesis succeeded in designing and implementing a flexible attribute component that provides the functionality for its employment in a graph transformation system.

It offers:

- Interactive and programmable access to attribute declarations and values;

  Apart from graphically modeling of systems, the attribute component can be used as a graphical database for information useful to the client.
- Ensuring consistency at all times;
- Support of different views;
- Attribute match checking during rule match search;
- Transformation of attributes;
- Rule variables that
  - are declared explicitly or implicitly through attributes,
  - are assigned values during a rule match
  - and are used during attribute transformation.
- Rule parameters that can be passed between rules and/or entered interactively "on demand";
- Application conditions;
- Connecting attribute objects through interfaces;
- Great extensibility concerning custom editors, additional attribute handlers and...

- Interpretation of Java expressions and thereby an enormous expressiveness.

The design addressed primarily a system for algebraic graph transformation. Nevertheless, its use is by no means limited to it. The only restriction is that the attributed objects must be typed.

New directions for development of the attribute component as well as the graph transformation system using it are:

- Distribution over networks, use as a Java applet;
- Thread safety;
- Generation of stand-alone prototypes from a graph grammar and the ability to be invoked from user code.

Appendix A

API - Application Programming Interface

This appendix presents all interfaces and some classes with their complete method specifications. The arrangement is according to the packages.

A.1 Interface Package agg.attribute

This package contains interfaces for the attribute tuples, contexts, the tuple observer mechanism and distribution.

A.1.1 Interface agg.attribute.AttrManager

public interface AttrManager

extends Serializable

Factory class interface for attribute-related objects; Provides creating services needed by graphical components.

Methods

- **newContext**

```
public abstract AttrContext newContext(int mapStyle)
```

Creating a new attribute context which is the root of a context tree;

Parameters: mapStyle - The kind of mapping that is done within this context; it is one of:

- `AttrMapping.PLAIN_MAP`: In Graph Transform.: rule mapping
- `AttrMapping.MATCH_MAP`: In Graph Transformation: matching

Returns: The new attribute root context

See Also: AttrContext (cf. Section A.1.12), AttrMapping (cf. Section A.1.13)

- **newContext**

```
public abstract AttrContext newContext(int mapStyle,
                                       AttrContext parent)
```

Creating a new attribute context which extends an existing one. In Graph Transformation, the setting of variables by matching corresponding graphical objects requires such a construction. It allows for keeping more that one rule match at a given time;

Parameters: mapStyle - The kind of mapping that is done within this context; it is one of:

- `AttrMapping.PLAIN_MAP`: In Graph Transform.: rule mapping
- `AttrMapping.MATCH_MAP`: In Graph Transformation: matching

parent - The context to extend

Returns: The new attribute context child

See Also: AttrContext (cf. Section A.1.12), AttrMapping (cf. Section A.1.13)

- **newLeftContext**

```
public abstract AttrContext newLeftContext(AttrContext context)
```

Creating a left rule side view for an existing rule context; Here, variables can be declared, but the assignment of complex expressions to single attribute values is forbidden.

Parameters: context - The context to generate the view on

Returns: The new attribute context view

See Also: AttrContext (cf. Section A.1.12)

- **newRightContext**

```
public abstract AttrContext newRightContext(AttrContext source)
```

Creating a view for an existing rule context, through which variables cannot be declared; complex expressions as attribute values are allowed, but only declared variables may be used.

Parameters: context - The context to generate the view on

Returns: The new attribute context view

See Also: AttrContext (cf. Section A.1.12)

- **getHandler**

```
public abstract AttrHandler getHandler(String name)
```

Getting an attribute handler by name.

Returns: The attribute handler.

- **getHandlers**

```
public abstract AttrHandler[] getHandlers()
```

Getting all attribute handlers that have been registered.

- **newType**

```
public abstract AttrType newType()
```

Creating a new attribute type.

Returns: The new attribute type

- **newInstance**

```
public abstract AttrInstance newInstance(AttrType type)
```

Creating a new attribute instance of the required type, without a context. Note that for such attributes, expressions must be constants. In Graph Transformation, it is used for creating a new attribute in the host graph.

Parameters: type - The type to use

Returns: The new attribute instance

- **newInstance**

```
public abstract AttrInstance newInstance(AttrType type,
                                         AttrContext context)
```

Creating a new attribute instance of the required type and in the given context or a context view. In Graph Transformation, it is used for creating a new attribute in a rule.

Parameters: type - The type to use context - The context to use

Returns: The new attribute instance

- **checkIfReadyToMatch**

```
public abstract void checkIfReadyToMatch(AttrContext ruleContext)

 throws AttrException
```

Checking if matching can be performed with respect to a given rule context. If the rule context in question is without inconsistencies, this method remains `silent`. Otherwise, it throws an exception whose message text describes the reason.

Throws: AttrException (cf. Section A.1.17) Describes reason for failure.

- **newMapping**

```
public abstract AttrMapping newMapping(AttrContext mappingContext,
                                       AttrInstance source,
                                       AttrInstance target)

 throws AttrMatchException
```

Mapping between two attribute instances; The mapping is done according to the context mapping property (match/plain) and is integrated into the context;

Parameters: mappingContext - The context to include the mapping in source - Mapping source attribute target - Mapping target attribute

Returns: A handle to the mapping; it can be used to undo the mapping (remove()) or to proceed to the next possible one (next()). If the mapping style for mappingContext is *MATCH_MAP*, a match is tried and necessary checks concerning non-injectiveness are performed. If this fails, *null* is returned.

Throws: AttrMatchException (cf. Section A.1.18) Describes reason for failure.

- **checkIfReadyToTransform**

```
public abstract void checkIfReadyToTransform(AttrContext matchContext)

 throws AttrException
```

Checking if a transformation can be performed with the attributes with respect to a given context. If the match context in question is complete and without inconsistencies, this method remains `silent`. Otherwise, it throws an exception whose message text describes the reason.

Throws: AttrException (cf. Section A.1.17) Describes reason for failure.

- **newViewSetting**

```
public abstract AttrViewSetting newViewSetting()
```

Creating a new mediator instance for loose coupling of attribute objects with their visual representation.

- **getDefaultOpenView**

```
public abstract AttrViewSetting getDefaultOpenView()
```

Obtaining the open view of the default view setting (`open` meaning: it considers permutations, but not hiding of members;).

- **getDefaultMaskedView**

```
public abstract AttrViewSetting getDefaultMaskedView()
```

Obtaining the masked view of the default view setting (`masked` meaning: it considers permutations as well as hiding of members;).

- **getDistributionBroker**

```
public abstract AttrDistributionBroker getDistributionBroker()
```

Returns the distribution broker.

A.1.2 Interface agg.attribute.AttrTuple

public interface AttrTuple

Methods

- **getAttrManager**

```
public abstract AttrManager getAttrManager()
```

Getting the attribute manager.

- **isValid**

```
public abstract boolean isValid()
```

Testing if the tuple is consistent and complete.

- **getMemberAt**

```
public abstract AttrMember getMemberAt(int index)
```

Getting a tuple member by its absolute (view-independent) index.

- **getMemberAt**

```
public abstract AttrMember getMemberAt(AttrViewSetting view,
                                       int index)
```

Getting a tuple member by its view-dependent index.

- **getMemberAt**

```
public abstract AttrMember getMemberAt(String name)
```

Getting a tuple member by its declaration name.

- **getIndexForMember**

```
public abstract int getIndexForMember(AttrMember m)
```

Translation between address- and number-oriented access.

Returns: The corresponding index if the member is within the tuple, -1 otherwise.

- **getNumberOfEntries**

```
public abstract int getNumberOfEntries()
```

Getting the absolute (view-independent) total number of entries (lines); The retrieval index range is [0 .. (getNumberOfEntries() - 1)].

Parameters: entryIndex - Index of entry.

- **getTypeAsString**

```
public abstract String getTypeAsString(int entryIndex)
```

Getting a view-independent representation of a type as String.

Parameters: entryIndex - Index of entry.

- **getNameAsString**

```
public abstract String getNameAsString(int entryIndex)
```

Getting a view-independent representation of a name as String.

Parameters: entryIndex - Index of entry.

- **getValueAsString**

```
public abstract String getValueAsString(int entryIndex)
```

Getting a view-independent representation of a value as String.

Parameters: entryIndex - Index of entry.

- **getNumberOfEntries**

```
public abstract int getNumberOfEntries(AttrViewSetting viewSetting)
```

Getting the view-dependent number of attribute entries (lines). The retrieval index range is [0 .. (getNumberOfEntries() - 1)].

Parameters: viewSetting - The view context which mandates how attribute tuples have to be represented.

- **getTypeAsString**

```
public abstract String getTypeAsString(AttrViewSetting viewSetting,
                                       int entryIndex)
```

Getting a view-dependent representation of a type as String.

Parameters: viewSetting - The view context which mandates how attribute tuples have to be represented. entryIndex - Index of entry.

- **getNameAsString**

```
public abstract String getNameAsString(AttrViewSetting viewSetting,
                                       int entryIndex)
```

Getting a view-dependent representation of a type as String.

Parameters: viewSetting - The view context which mandates how attribute tuples have to be represented. entryIndex - Index of entry.

- **getValueAsString**

```
public abstract String getValueAsString(AttrViewSetting viewSetting,
                                        int entryIndex)
```

Getting a view-dependent representation of a type as String.

Parameters: viewSetting - The view context which mandates how attribute tuples have to be represented. entryIndex - Index of entry.

- **addObserver**

```
public abstract void addObserver(AttrObserver attrObs)
```

Adding a new attribute observer.

Parameters: attrObs The - attribute observer to be registered.

- **removeObserver**

```
public abstract void removeObserver(AttrObserver attrObs)
```

Removing an attribute observer from the list of observers.

Parameters: attrObs - The attribute observer to be registered.

A.1.3 Interface agg.attribute.AttrMember

public interface AttrMember

extends Serializable

An abstract tuple member interface.

Methods

- **isValid**

```
public abstract boolean isValid()
```

Testing if the member is consistent and complete.

- **getValidityReport**

```
public abstract String getValidityReport()
```

Returns a text describing the errors in this member, or null if the member is correct.

- **getHoldingTuple**

```
public abstract AttrTuple getHoldingTuple()
```

Returns the tuple that contains this member.

- **getName**

```
public abstract String getName()
```

Returns the member name.

- **getIndexInTuple**

```
public abstract int getIndexInTuple()
```

Returns the member index within the containing tuple.

A.1.4 Interface agg.attribute.AttrType

public interface AttrType

extends Serializable, AttrTuple (cf. Section A.1.2)

A tuple of declarations each consisting of a type and name.

Methods

- **addMember**

```
public abstract AttrTypeMember addMember(AttrHandler handler,
                                         String type,
                                         String name)
```

Adding a declaration.

Parameters: handler - attribute handler for the entry type; type - textual representation of the entry type; name - name (selector) of the entry within the attribute tuple.

Returns: The newly created member declaration.

- **addMember**

```
public abstract AttrTypeMember addMember()
```

Adding an empty declaration. The new declaration member is returned and can be extended by calling the respective AttrTypeMember methods.

- **deleteMemberAt**

```
public abstract void deleteMemberAt(String name)
```

Delete a declaration.

Parameters: name - name (selector) of the entry within the attribute tuple.

- **deleteMemberAt**

```
public abstract void deleteMemberAt(int index)
```

Delete a declaration.

Parameters: index - index of the member within the attribute tuple.

- **deleteMemberAt**

```
public abstract void deleteMemberAt(AttrViewSetting viewSetting,
                                    int slot)
```

Delete a declaration.

Parameters: viewSetting - view setting to relate to. slot - slot of the member within the view of the attribute tuple.

- **addEntry**

```
public abstract void addEntry(AttrHandler handler,
                              String type,
                              String name)
```

Note: addEntry() is deprecated.

Adding a declaration.

Parameters: handler - attribute handler for the entry type; type - textual representation of the entry type; name - name (selector) of the entry within the attribute tuple.

See Also: addMember (cf. Section A.1.4)

- **deleteEntry**

```
public abstract void deleteEntry(String name)
```

Note: deleteEntry() is deprecated.

Delete a declaration.

Parameters: name - name (selector) of the entry within the attribute tuple.

See Also: deleteMemberAt

A.1.5 Interface agg.attribute.AttrTypeMember

public interface AttrTypeMember

extends AttrMember (cf. Section A.1.3)

The interface for a member of an attribute type.

Methods

- **delete**

```
public abstract void delete()
```

Removes itself from the tuple.

- **getName**

```
public abstract String getName()
```

Retrieving its name.

- **setName**

```
public abstract void setName(String name)
```

Setting a name.

- **getType**

```
public abstract HandlerType getType()
```

Retrieving its type. Returns null if no type is set or if the type is not valid.

- **getTypeName**

```
public abstract String getTypeName()
```

Retrieving its type name as string.

- **setType**

```
public abstract void setType(String typeName)
```

Setting its type.

- **getHandler**

```
public abstract AttrHandler getHandler()
```

Retrieving its attribute handler.

- **setHandler**

```
public abstract void setHandler(AttrHandler h)
```

Setting its attribute handler.

A.1.6 Interface agg.attribute.AttrInstance

public interface AttrInstance

extends Serializable, AttrTuple (cf. Section A.1.2)

Interface of tuples of attribute values.

Methods

- **getType**

```
public abstract AttrType getType()
```

Retrieving the type of an instance.

- **getContext**

```
public abstract AttrContext getContext()
```

Retrieving the context of an instance.

- **isValueSetAt**

```
public abstract boolean isValueSetAt(String name)
```

Test, if a value is set or not.

- **getValueAt**

```
public abstract Object getValueAt(String name)
```

Retrieving the value of an entry. If the result is `null`, the reason can be: 1. The value is set as `null`; 2. The value is not set at all. For testing, if the value was set as `null` or not set at all, use `isValueSetAt()` of this class.

- **setValueAt**

```
public abstract void setValueAt(Object value,
                                String name)
```

Setting the value of an entry directly.

Parameters: value - Any object instance. name - specifies the entry to change.

- **setExprValueAt**

```
public abstract void setExprValueAt(String expr,
                                    String name)
```

Evaluating an expression and setting its value as an entry.

Parameters: expr - textual expression representation; name - specifies the entry to change.

- **setExprAt**

```
public abstract void setExprAt(String expr,
                               String name)
```

Setting an expression as an entry without immediate evaluation. Syntax and type checking are performed.

Parameters: expr - textual expression representation; name - specifies the entry to change;

- **copy**

```
public abstract void copy(AttrInstance source)
```

Copying the contents of an attribute instance into another.

- **getNumberOfFreeVariables**

```
public abstract int getNumberOfFreeVariables(AttrContext context)
```

Getting the number of variables declared by this instance which have no value assigned to them yet. Each variable name is counted only once, even if it is used more than once in this tuple.

Returns: The number of free variables.

- **apply**

```
public abstract void apply(AttrInstance rightSide,
                           AttrContext context)
```

Applying a rule; the substitutions occur *in-place*, i.e. in the recipient; In Graph Transformation, this method is applied to attributes of host graph objects, *rightSide* being an attribute of the right side of the rule and *context* being the *match*-context built up by subsequently matching the attributes of corresponding graphical objects.

A.1.7 Interface agg.attribute.AttrInstanceMember

public interface AttrInstanceMember

extends AttrMember (cf. Section A.1.3)

The interface for an instance tuple member.

Methods

- **getDeclaration**

```
public abstract AttrTypeMember getDeclaration()
```

Retrieving the type.

- **isSet**

  `public abstract boolean isSet()`

  Test, if the value is set or not.

- **getExpr**

  `public abstract HandlerExpr getExpr()`

  Retrieving the expression (value) contained in this member. The result can be queried and set according to the agg.attribute.handler.HandlerExpr interface.

- **setExpr**

  `public abstract void setExpr(HandlerExpr expr)`

  Setting the expression (value) contained in this member.

- **getExprAsObject**

  `public abstract Object getExprAsObject()`

  Retrieving the value of an entry. If the result is `null`, the reason can be: 1. The value is set as `null`; 2. The value is not set at all. For testing if the value was set as `null` or not set at all, use `isSet()` of this interface.

- **getExprAsText**

  `public abstract String getExprAsText()`

  Returns the textual representation of the expression.

- **setExprAsObject**

  `public abstract void setExprAsObject(Object value)`

  Setting the value of an instance member directly.

  Parameters: value - Any object instance.

- **setExprAsEvaluatedText**

```
public abstract void setExprAsEvaluatedText(String expr)
```

Evaluating an expression and setting its value as this member's entry.

Parameters: expr - textual expression representation;

- **setExprAsText**

```
public abstract void setExprAsText(String expr)
```

Setting an expression for this member without immediate evaluation. Syntax and type checking are performed.

Parameters: expr - textual expression representation;

A.1.8 Interface agg.attribute.AttrConditionTuple

public interface AttrConditionTuple

extends AttrInstance (cf. Section A.1.6)

The interface for a condition tuple.

Methods

- **addCondition**

```
public abstract AttrConditionMember addCondition(String expr)
```

Adding of a condition member, returning the member. For deletion,

See Also: AttrConditionMember (cf. Section A.1.9)

- **isDefinite**

```
public abstract boolean isDefinite()
```

Test, if all members can yield true or false.

- **isTrue**

  `public abstract boolean isTrue()`

  Test, if ANDing of all members yields true.

- **isFalse**

  `public abstract boolean isFalse()`

  Test, if the tuple contains members which can be evaluated and yield `false`.

A.1.9 Interface agg.attribute.AttrConditionMember

public interface AttrConditionMember

extends AttrInstanceMember (cf. Section A.1.7)

The interface for an instance tuple member.

Methods

- **delete**

  `public abstract void delete()`

  Removes this member from its tuple.

- **isDefinite**

  `public abstract boolean isDefinite()`

  Test, if the expression can yield true or false.

- **isTrue**

  `public abstract boolean isTrue()`

  Test, if the expression yields true.

- **isFalse**

  `public abstract boolean isFalse()`

  Test, if the expression yields false.

A.1.10 Interface agg.attribute.AttrVariableTuple

public interface AttrVariableTuple

extends AttrInstance (cf. Section A.1.6)

The interface for a tuple of variables.

Methods

- **isDefinite**

  `public abstract boolean isDefinite()`

  Test, if all variables evaluate to definite values.

A.1.11 Interface agg.attribute.AttrVariableMember

public interface AttrVariableMember

extends AttrInstanceMember (cf. Section A.1.7)

The interface for a variable tuple member.

Methods

- **delete**

  `public abstract void delete()`

  Removes this member from its tuple.

- **isDefinite**

  `public abstract boolean isDefinite()`

  Test, if the variable evaluates to a definite value.

- **isInputParameter**

  `public abstract boolean isInputParameter()`

Tests if this variable is an IN-parameter.

- **setInputParameter**

```
public abstract void setInputParameter(boolean b)
```

Sets, if the variable is to be an IN-parameter.

- **isOutputParameter**

```
public abstract boolean isOutputParameter()
```

Tests if this variable is an OUT-parameter.

- **setOutputParameter**

```
public abstract void setOutputParameter(boolean b)
```

Sets, if the variable is to be an OUT-parameter.

A.1.12 Interface agg.attribute.AttrContext

public interface AttrContext

extends Serializable, SymbolTable (cf. Section A.5.4)

Framework for allocation of variables, administration of attribute mappings and application conditions in rules.

Methods

- **getConditions**

```
public abstract AttrConditionTuple getConditions()
```

Returns the application conditions tuple of this context.

- **getVariables**

```
public abstract AttrVariableTuple getVariables()
```

Returns the variable tuple of this context.

- **doesAllowComplexExpressions**

```
public abstract boolean doesAllowComplexExpressions()
```

Checks if complex expressions are permitted in this context.

- **doesAllowNewVariables**

```
public abstract boolean doesAllowNewVariables()
```

Checks if implicit variable declarations are permitted in this context.

- **doesAllowEmptyValues**

```
public abstract boolean doesAllowEmptyValues()
```

Checks if empty attribute values are permitted in this context.

- **setAllowVarDeclarations**

```
public abstract void setAllowVarDeclarations(boolean isAllowed)
```

Sets if implicit variable declarations are permitted in this context.

- **setAllowComplexExpr**

```
public abstract void setAllowComplexExpr(boolean isAllowed)
```

Sets if complex expressions are permitted in this context.

- **setAllowEmptyValues**

```
public abstract void setAllowEmptyValues(boolean isAllowed)
```

Sets if empty attribute values are permitted in this context.

- **freeze**

```
public abstract void freeze()
```

Switching on the freeze mode; mapping removals are deferred until `defreeze()` is called.

- **defreeze**

```
public abstract void defreeze()
```

Perform mapping removals which were delayed during the freeze mode.

- **getType**

```
public abstract HandlerType getType(String name)
```

Getting the type of an identifier. getType(String) and getExpr(String) allow to use an AttrContext as a SymbolTable when using an AttrHandler.

Parameters: name - Identifier's name

Returns: Identifier's type

- **getExpr**

```
public abstract HandlerExpr getExpr(String name)
```

Getting the value of an identifier. getType(String) and getExpr(String) allow to use an AttrContext as a SymbolTable when using an AttrHandler.

Parameters: name - Identifier's name

Returns: Identifier's value as expression

A.1.13 Interface agg.attribute.AttrMapping

public interface AttrMapping

extends Serializable

Variables

- **PLAIN_MAP**

```
public static final int PLAIN_MAP
```

Constant for the *plain* mapping mode. In Graph Transformation this stands for a mapping as in the rule morphisms.

- **MATCH_MAP**

```
public static final int MATCH_MAP
```

Constant for the *match* mapping mode. In Graph Transformation this stands for a mapping as in match constructions.

Methods

- **next**

```
public abstract boolean next()
```

Use the next possible mapping;

Returns: *true* if more subsequent mappings exist, *false* otherwise.

- **remove**

```
public abstract void remove()
```

Discard mapping; Removes variable assignments made by this mapping from its context and dissolves the connection between the attribute instances.

A.1.14 Interface agg.attribute.AttrObserver

public interface AttrObserver

extends Serializable

Methods

- **attributeChanged**

```
public abstract void attributeChanged(AttrEvent event)
```

- **isPersistentFor**

```
public abstract boolean isPersistentFor(AttrTuple at)
```

A.1.15 Interface agg.attribute.AttrDistributionBroker

public interface AttrDistributionBroker

extends Serializable

Mediator interface for distribution purposes. Provides services for creating and maintaining of interface/local-relations between attribute tuples and contexts.

Methods

- **connect**

```
public abstract void connect(AttrType interfaceType,
                             AttrType localType)
```

Makes a type tuple into an interface of another type tuple.

- **disconnect**

```
public abstract void disconnect(AttrType interfaceType,
                                AttrType localType)
```

Ends a type tuple's role as an interface of another type tuple.

- **connect**

```
public abstract void connect(AttrInstance interfaceInstance,
                             AttrInstance localInstance)
```

Makes an instance tuple into an interface of another interface tuple.

• **disconnect**

```
public abstract void disconnect(AttrInstance interfaceInstance,
                                AttrInstance localInstance)
```

Ends an instance tuple's role as an interface of another interface tuple.

• **connect**

```
public abstract void connect(AttrContext interfaceContext,
                             AttrContext localContext)
```

Makes a context into an interface of another context.

• **disconnect**

```
public abstract void disconnect(AttrContext interfaceContext,
                                AttrContext localContext)
```

Ends a context's role as an interface of another context.

A.1.16 Interface agg.attribute.AttrEvent

public interface AttrEvent

Attribute event interface for delivering information about attribute changes to clients.

See Also: AttrObserver (cf. Section A.1.14), AttrTuple (cf. Section A.1.2)

Variables

• **GENERAL_CHANGE**

```
public static final int GENERAL_CHANGE
```

Change not specified. (The event receiver should update the whole tuple

• **MEMBER_ADDED**

```
public static final int MEMBER_ADDED
```

A new member was added.

- **MEMBER_DELETED**

  `public static final int MEMBER_DELETED`

  A member was deleted.

- **MEMBER_MODIFIED**

  `public static final int MEMBER_MODIFIED`

  A member was modified, no further specification.

- **MEMBER_RENAMED**

  `public static final int MEMBER_RENAMED`

  A member was renamed.

- **MEMBER_RETYPED**

  `public static final int MEMBER_RETYPED`

  A member was retyped.

- **MEMBER_VALUE_MODIFIED**

  `public static final int MEMBER_VALUE_MODIFIED`

  The value of an attribute was modified.

- **MEMBER_VALUE_CORRECTNESS**

  `public static final int MEMBER_VALUE_CORRECTNESS`

  The state of correctness of a member value has changed.

- **ATTR_EVENT_MAX_ID**

  `public static final int ATTR_EVENT_MAX_ID`

  The highest id value for this interface. Extending interfaces must not have id costants below this value.

Methods

- **getSource**

  ```
  public abstract AttrTuple getSource()
  ```

  Getting the originator of the event.

- **getID**

  ```
  public abstract int getID()
  ```

  Getting the message id.

- **getIndex**

  ```
  public abstract int getIndex()
  ```

  Getting the first position index.

- **getIndex0**

  ```
  public abstract int getIndex0()
  ```

  Getting the first position index.

- **getIndex1**

  ```
  public abstract int getIndex1()
  ```

  Getting the second position index.

A.1.17 Class agg.attribute.AttrException

```
java.lang.Object
 |
 +----java.lang.Throwable
         |
         +----java.lang.Exception
                 |
                 +----agg.attribute.AttrException
```

public class AttrException

extends Exception

Constructors

- **AttrException**

```
public AttrException(String msg)
```

A.1.18 Class agg.attribute.AttrMatchException

```
java.lang.Object
 |
 +----java.lang.Throwable
         |
         +----java.lang.Exception
                 |
                 +----agg.attribute.AttrException
                         |
                         +----agg.attribute.AttrMatchException
```

public class AttrMatchException

extends AttrException (cf. Section A.1.17)

Variables

- **SIMPLE**

```
public static final int SIMPLE
```

- **VARIABLE_BINDING**

```
public static final int VARIABLE_BINDING
```

- **id**

```
protected int id
```

- **firstBindingTuple**

```
protected AttrInstance firstBindingTuple
```

Constructors

- **AttrMatchException**

```
public AttrMatchException(String msg,
                          int id)
```

- **AttrMatchException**

```
public AttrMatchException(String msg,
                          int id,
                          AttrInstance firstBindingTuple)
```

Methods

- **getID**

```
public int getID()
```

- **getFirstBindingTuple**

```
public AttrInstance getFirstBindingTuple()
```

A.2 Interface Package agg.attribute.view

This package contains interfaces for the attribute view.

A.2.1 Interface agg.attribute.view.AttrViewSetting

public interface AttrViewSetting

extends Serializable

Mediator interface, facilitating view-dependent access to attribute objects. The *Mediator* design pattern was chosen for a loose and lightweight coupling of attribute objects and their visual representation. It also allows view-dependent (editor) and view-independent (graph transformation unit) users of the attribute component to identify their attribute objects by the same handles. Please pay attention that the integer selectors for attribute tuple members are not absolute indexes as in the AttrTuple interface. Rather, they are *slots*, member positions in respect to this view. There can be an arbitrary number of views, each holding exactly one (changeable) representation of an attribute tuple (order of members, hiding of members). Each view has two `subviews`, an `open view` and a `masked view`. They basically share the same tuple layout information, with one exception. Tuple access using the subview obtained by calling getOpenView() does not hide the `hidden` members, although they can be hidden by invoking setVisibleAt(aTuple, false, aSlot). The hiding effect only occurs when using the subview obtained by calling getMaskedView(). The order of members is consistent, i.e. if `memberA == aTuple.getMember( slotA1, aViewSetting.getOpenView()) && mamberB == aTuple.getMember( slotB1, aViewSetting.getOpenView()) && memberA == aTuple.getMember( slotA2, aViewSetting.getMaskedView()) && memberB == aTuple.getMember( slotB2, aViewSetting.getMaskedView()) && slotA1 then` slotA2 The event indices (slots) are delivered according to the view for which the observer registered. When accessing a tuple using these slots always remember to supply the appropriate view. It can be obtained by invoking getView() on the delivered event.

Methods

- **getOpenView**

```
public abstract AttrViewSetting getOpenView()
```

Returns the `open subview`, manipulating of visibility of members (setVisibleAt(...)) only affects the other subview.

See Also: getMaskedView (cf. Section A.2.1)

- **getMaskedView**

```
public abstract AttrViewSetting getMaskedView()
```

Returns the `masked subview`, manipulating of visibility of members (setVisibleAt(...)) affects this subview.

See Also: getOpenView (cf. Section A.2.1)

- **addObserver**

```
public abstract void addObserver(AttrViewObserver o,
                                 AttrTuple attr)
```

Adding an observer for an attribute tuple's representation.

- **removeObserver**

```
public abstract void removeObserver(AttrViewObserver o,
                                    AttrTuple attr)
```

Removing an observer for an attribute tuple's representation.

- **convertIndexToSlot**

```
public abstract int convertIndexToSlot(AttrTuple attr,
                                       int index)
```

Returns the slot position in the view layout for `attr` at `index`.

- **convertSlotToIndex**

```
public abstract int convertSlotToIndex(AttrTuple attr,
                                       int slot)
```

Returns the index for `attr` at `slot`, as set in this view layout.

- **getSize**

```
public abstract int getSize(AttrTuple attr)
```

Returns the number of members that are visible in this view.

- **isVisible**

```
public abstract boolean isVisible(AttrTuple attr,
                                  int slot)
```

Testing if the attribute member at the specified slot is visible in this view.

- **setVisibleAt**

```
public abstract void setVisibleAt(AttrTuple attr,
                                  boolean b,
                                  int slot)
```

Setting, if the attribute member at the specified slot of this view should be visible or not.

- **setAllVisible**

```
public abstract void setAllVisible(AttrTuple attr,
                                   boolean b)
```

Setting, if all attribute members of `attr` should either be at once made visible or hidden.

- **moveSlotInserting**

```
public abstract void moveSlotInserting(AttrTuple attr,
                                       int srcSlot,
                                       int destSlot)
```

Moves the member at *srcSlot* to *destSlot*, inserting-wise.

- **moveSlotAppending**

```
public abstract void moveSlotAppending(AttrTuple attr,
                                       int srcSlot,
                                       int destSlot)
```

Moves the member at *srcSlot* to *destSlot*, appending-wise.

- **resetTuple**

```
public abstract void resetTuple(AttrTuple attr)
```

Reset the tuple layout, so each slot number is the same as the index it contains, with all slots visible.

A.2.2 Interface agg.attribute.view.AttrViewObserver

public interface AttrViewObserver

Methods

- **attributeChanged**

```
public abstract void attributeChanged(AttrViewEvent event)
```

A.2.3 Interface agg.attribute.view.AttrViewEvent

public interface AttrViewEvent

extends AttrEvent (cf. Section A.1.16)

Attribute event interface for delivering information about attribute changes to clients.

See Also: AttrObserver (cf. Section A.1.14), AttrTuple (cf. Section A.1.2)

Variables

- **MEMBER_MOVED**

```
public static final int MEMBER_MOVED
```

A member was moved.

- **MEMBER_VISIBILITY**

```
public static final int MEMBER_VISIBILITY
```

A member was shown or hidden .

Methods

- **getView**

```
public abstract AttrViewSetting getView()
```

Getting the view setting.

A.3 Interface Package agg.attribute.gui

This package contains interfaces for editors of attribute tuples, contexts and the attribute manager customization.

A.3.1 Interface agg.attribute.gui.AttrEditorManager

public interface AttrEditorManager

Abstract factory for attribute editors.

Methods

- **getTopEditor**

```
public abstract AttrTopEditor getTopEditor(AttrManager m,
                                           AttrViewSetting v)
```

Creates a comprehensive editor which allows to manipulate an attribute context, an attribute tuple (instance and its type) and to customize the options of the attribute manager and handlers.

- **getSmallEditorForInstance**

```
public abstract AttrTupleEditor getSmallEditorForInstance(AttrManager m,
                                                          AttrViewSetting v,
                                                          AttrInstance inst)
```

Creates a new small editor for an attribute tuple instance, where only the values can be changed. The editor window component can be integrated into graphical objects which hold the attributes.

- **getHandlerEditorManager**

```
public abstract HandlerEditorManager getHandlerEditorManager()
```

Getting the manager (abstract factory) for handler editors.

A.3.2 Interface agg.attribute.gui.AttrEditor

public interface AttrEditor

extends PropertyEditor

Abstract interface, parent of all attribute editors. it provides basic operations like setting and getting of the attribute and the editor manager.

Methods

- **getComponent**

```
public abstract Component getComponent()
```

Getting the AWT component (window entity) that can be integrated in a larger GUI context or diplayed separately.

- **getAttrManager**

```
public abstract AttrManager getAttrManager()
```

Getting the attribute manager.

- **setAttrManager**

```
public abstract void setAttrManager(AttrManager m)
```

Setting the attribute manager.

- **getEditorManager**

```
public abstract AttrEditorManager getEditorManager()
```

Getting the editor manager.

- **setEditorManager**

```
public abstract void setEditorManager(AttrEditorManager m)
```

Setting the editor manager.

A.3.3 Interface agg.attribute.gui.AttrTupleEditor

public interface AttrTupleEditor

extends AttrEditor (cf. Section A.3.2)

Editor for an attribute tuple. Some instances thereof can be restricted to attribute tuple instances (AttrInstance), others can have manipulative access to the tuple type as well.

Methods

- **setTuple**

  ```
  public abstract void setTuple(AttrTuple anAttrTuple)
  ```

  Setting the attribute tuple to be edited.

- **getTuple**

  ```
  public abstract AttrTuple getTuple()
  ```

  Getting the edited attribute tuple.

- **setViewSetting**

  ```
  public abstract void setViewSetting(AttrViewSetting anAttrViewSetting)
  ```

  Setting the view for the editor.

- **getViewSetting**

  ```
  public abstract AttrViewSetting getViewSetting()
  ```

  Getting the editor's view.

A.3.4 Interface agg.attribute.gui.AttrContextEditor

public interface AttrContextEditor

extends AttrEditor (cf. Section A.3.2)

Editor for an attribute context, allows editing of variables and application conditions of a (rule/match) context.

Methods

- **setContext**

```
public abstract void setContext(AttrContext anAttrContext)
```

Setting the edited context.

- **getContext**

```
public abstract AttrContext getContext()
```

Getting the edited context.

A.3.5 Interface agg.attribute.gui.AttrCustomizingEditor

public interface AttrCustomizingEditor

extends AttrEditor (cf. Section A.3.2)

Customizing of an attribute manager as well as his handlers.

A.3.6 Interface agg.attribute.gui.AttrTopEditor

public interface AttrTopEditor

extends AttrContextEditor, AttrTupleEditor, AttrCustomizingEditor (cf. Section A.3.5)

Most comprehensive editor for the attribute component. A comprehensive editor which allows to manipulate an attribute context, an attribute tuple (instance and its type) and to customize the options of the attribute manager and handlers.

A.4 Interface Package agg.attribute.facade

This package contains interfaces for facades.

A.4.1 Interface agg.attribute.facade.EditorFacade

public interface EditorFacade

Methods

- **getAttrManager**

  ```
  public abstract AttrManager getAttrManager()
  ```

  Returns the default attribute manager.

- **getTopEditor**

  ```
  public abstract AttrTopEditor getTopEditor()
  ```

  Returns a comprehensive editor, allowing to edit: - an attribute tuple, including its type member properties, such as the member handlers, types and names; the hiding and moving of members is possible; - a rule or match context, i.e. its variables and application conditions; - options of the attribute component (customization).

- **getSmallEditorForInstance**

  ```
  public abstract AttrTupleEditor getSmallEditorForInstance(AttrInstance inst)
  ```

  Returns a compact editor for an attribute tuple instance, showing the members' types, names and the instance members, where only the latter can be changed.

- **editInstance**

  ```
  public abstract void editInstance(AttrTupleEditor ed,
                                    AttrInstance inst)
  ```

  Setting (*loading*) an attribute tuple instance into an attribute tuple editor (or into a *top editor*, which is a subclass thereof.

- **editContext**

  ```
  public abstract void editContext(AttrContextEditor ed,
                                   AttrContext ctx)
  ```

  Setting (*loading*) an attribute (rule or match) context into a *top editor*.

A.4.2 Interface agg.attribute.facade.InformationFacade

public interface InformationFacade

Collection of methods for storing and retrieving information in attribute tuples and members.

Methods

- **getAttrManager**

```
public abstract AttrManager getAttrManager()
```

Returns the default attribute manager which can be used for advanced operations not provided by this facade.

- **getJavaHandler**

```
public abstract AttrHandler getJavaHandler()
```

Returns the java expression handler. This can then be used for creating a tuple type member.

- **addObserver**

```
public abstract void addObserver(AttrTuple tuple,
                                 AttrObserver attrObs)
```

Adding a new attribute observer.

Parameters: tuple - The attribute tuple (type or instance) to observe. attrObs - The attribute observer to be registered.

- **removeObserver**

```
public abstract void removeObserver(AttrTuple tuple,
                                    AttrObserver attrObs)
```

Removing an attribute observer from the list of observers.

Parameters: tuple - The attribute tuple (type or instance) observed. attrObs - The attribute observer to be registered.

- **createTupleType**

```
public abstract AttrType createTupleType()
```

Returns a new attribute tuple type, using the default attribute manager.

- **addMember**

```
public abstract AttrTypeMember addMember(AttrType tupleType,
                                         AttrHandler handler,
                                         String memberType,
                                         String name)
```

Adding a member declaration to a tuple type.

Parameters: tupleType - the tuple type to be extended. handler - attribute handler for the entry type; memberType - textual representation of the member type; name - name (selector) of the entry within the attribute tuple. The new declaration member is returned and can be extended by calling the respective AttrTypeMember methods.

- **addMember**

```
public abstract AttrTypeMember addMember(AttrType tupleType)
```

Adding an empty member declaration to a tuple type.

Parameters: tupleType - the tuple type to be extended. The new declaration member is returned and can be extended by calling the respective AttrTypeMember methods.

- **deleteMemberAt**

```
public abstract void deleteMemberAt(AttrType tupleType,
                                    String name)
```

Delete a member declaration from a tuple type.

Parameters: tupleType - the tuple type. name - name (selector) of the entry within the attribute tuple.

- **deleteMemberAt**

```
public abstract void deleteMemberAt(AttrType tupleType,
                                    int index)
```

Delete a member declaration from a tuple type.

Parameters: tupleType - the tuple type. index - index of the member within the attribute tuple.

- **getTypeMemberAt**

```
public abstract AttrTypeMember getTypeMemberAt(AttrType tupleType,
                                               int index)
```

Getting a tuple type member by its absolute (view-independent) index.

- **getTypeMemberAt**

```
public abstract AttrTypeMember getTypeMemberAt(AttrType tupleType,
                                               String name)
```

Getting a tuple type member by its declaration name.

- **getName**

```
public abstract String getName(AttrTypeMember memberDecl)
```

Retrieving the member name.

- **setName**

```
public abstract void setName(AttrTypeMember memberDecl,
                             String memberName)
```

Setting a member type name.

- **getType**

```
public abstract HandlerType getType(AttrTypeMember memberDecl)
```

Retrieving the type. Returns null if no type is set or if the type is not valid.

- **getTypeName**

```
public abstract String getTypeName(AttrTypeMember memberDecl)
```

Retrieving the member type name as string.

- **setType**

```
public abstract void setType(AttrTypeMember memberDecl,
                             String typeName)
```

Setting the member type.

- **getHandler**

```
public abstract AttrHandler getHandler(AttrTypeMember memberDecl)
```

Retrieving the member attribute handler.

- **setHandler**

```
public abstract void setHandler(AttrTypeMember memberDecl,
                                AttrHandler h)
```

Setting the member attribute handler.

- **createTupleInstance**

```
public abstract AttrInstance createTupleInstance(AttrType type,
                                                 AttrContext context)
```

Creating a new attribute instance of the required type and in the given context or a context view. In Graph Transformation, it is used for creating a new attribute in a rule.

Parameters: type - The type to use context - The context to use, can be null

Returns: The new attribute instance

- **getInstanceMemberAt**

```
public abstract AttrInstanceMember getInstanceMemberAt(AttrInstance tupleInstance,
                                                       int index)
```

Getting a tuple instance member by its absolute (view-independent) index.

- **getInstanceMemberAt**

```
public abstract AttrInstanceMember getInstanceMemberAt(AttrInstance tupleInstance,
                                                       String name)
```

Getting a tuple instance member by its declaration name.

- **getDeclaration**

```
public abstract AttrTypeMember getDeclaration(AttrInstanceMember instanceMember)
```

Retrieving an instance member's type.

- **isSet**

```
public abstract boolean isSet(AttrInstanceMember instanceMember)
```

Test, if the member value is set or not.

- **getExpr**

```
public abstract HandlerExpr getExpr(AttrInstanceMember instanceMember)
```

Retrieving the expression (value) contained in a member. The result can be queried and set according to the agg.attribute.handler.HandlerExpr interface.

- **setExpr**

```
public abstract void setExpr(AttrInstanceMember instanceMember,
                             HandlerExpr expr)
```

Setting the expression (value) contained in this member.

- **getExprAsObject**

```
public abstract Object getExprAsObject(AttrInstanceMember instanceMember)
```

Retrieving the value of a member. If the result is `null`, the reason can be: 1. The value is set as `null`; 2. The value is not set at all. For testing if the value was set as `null` or not set at all, use `isSet()` of this interface.

- **setExprAsObject**

```
public abstract void setExprAsObject(AttrInstanceMember instanceMember,
                                     Object value)
```

Setting the value of an instance member directly.

Parameters: instanceMember - The member of an attribute tuple instance. value - Any object instance.

- **getExprAsText**

```
public abstract String getExprAsText(AttrInstanceMember instanceMember)
```

Returns the textual representation of a member's expression.

- **setExprAsEvaluatedText**

```
public abstract void setExprAsEvaluatedText(AttrInstanceMember instanceMember,
                                            String expr)
```

Evaluating an expression and setting its value as a member's entry.

Parameters: instanceMember - The member of an attribute tuple instance. expr - textual expression representation;

- **setExprAsText**

```
public abstract void setExprAsText(AttrInstanceMember instanceMember,
                                   String expr)
```

Setting an expression for a member without immediate evaluation. Syntax and type checking are performed.

Parameters: instanceMember - The member of an attribute tuple instance. expr - textual expression representation;

A.4.3 Interface agg.attribute.facade.TransformationFacade

public interface TransformationFacade

Methods

- **getAttrManager**

```
public abstract AttrManager getAttrManager()
```

Returns the default attribute manager which can be used for advanced operations not provided by this facade.

- **newRuleContext**

```
public abstract AttrContext newRuleContext()
```

Creates and returns a new rule context. Typically calls newContext(AttrMapping.PLAIN_MAP) of the default AttrManager.

- **newNegativeRuleContext**

```
public abstract AttrContext newNegativeRuleContext(AttrContext ruleContext)
```

Creates and returns a new negative application condition context. Typically calls newContext(AttrMapping.PLAIN_MAP, ruleContext) of the default AttrManager.

- **getLeftContext**

```
public abstract AttrContext getLeftContext(AttrContext ruleContext)
```

Returns the left side context to a rule context. Typically calls newLeftContext(ruleContext) of the default AttrManager.

- **getRightContext**

```
public abstract AttrContext getRightContext(AttrContext ruleContext)
```

Returns the right side context to a rule context. Typically calls newRightContext(ruleContext) of the default AttrManager.

- **checkIfReadyToMatch**

```
public abstract void checkIfReadyToMatch(AttrContext ruleContext)

 throws AttrException
```

Checking if matching can be performed with respect to a given rule context. If the rule context in question is without inconsistencies, this method remains `silent`. Otherwise, it throws an exception whose message text describes the reason.

Throws: AttrException (cf. Section A.1.17) Describes reason for failure.

- **newMatchContext**

```
public abstract AttrContext newMatchContext(AttrContext ruleContext)
```

Creates and returns a new match context to a rule context. Typically calls newContext(AttrMapping.MATCH_MAP, ruleContext) of the default AttrManager.

- **newNegativeMatchContext**

```
public abstract AttrContext newNegativeMatchContext(AttrContext negRuleContext,
                                                    AttrContext matchContext)
```

Creates and returns a new match context to a negative rule context. Typically calls newContext(AttrMapping.MATCH_MAP, negRuleContext) of the default AttrManager and copies the variable assignments from matchContext.

- **newMapping**

```
public abstract AttrMapping newMapping(AttrContext mappingContext,
                                       AttrInstance source,
                                       AttrInstance target)

 throws AttrMatchException
```

Mapping between two attribute instances; The mapping is done according to the context property (rule/match) and is integrated into the context;

Parameters: mappingContext - The context to include the mapping in source - Mapping source attribute target - Mapping target attribute

Returns: A handle to the mapping; it can be used to undo the mapping (remove()) or to proceed to the next possible one (next()). If the mapping context is that of a match, a match on the attributes is tried. If this fails, *null* is returned.

Throws: AttrMatchException (cf. Section A.1.18) Describes reason for failure.

- **nextMapping**

```
public abstract boolean nextMapping(AttrMapping mapping)
```

Use the next possible attribute mapping;

Returns: *true* if more subsequent mappings exist, *false* otherwise.

- **removeMapping**

```
public abstract void removeMapping(AttrMapping mapping)
```

Discard mapping; Removes variable assignments made by a mapping from its context and dissolves the connection between the attribute instances.

- **checkIfReadyToTransform**

```
public abstract void checkIfReadyToTransform(AttrContext matchContext)

 throws AttrException
```

Checking if a transformation can be performed with the attributes with respect to a given context. If the match context in question is complete and without inconsistencies, this method remains `silent`. Otherwise, it throws an exception whose message text describes the reason.

Throws: AttrException (cf. Section A.1.17) Describes reason for failure.

- **getNumberOfFreeVariables**

```
public abstract int getNumberOfFreeVariables(AttrInstance tuple,
                                              AttrContext context)
```

Getting the number of variables declared by an instance which have no value assigned to them yet. Each variable name is counted only once, even if it is used more than once in this tuple.

Returns: The number of free variables.

- **apply**

```
public abstract void apply(AttrInstance workGraphInst,
                           AttrInstance rightSideInst,
                           AttrContext context)
```

Applying a rule; the substitutions occur *in-place*, i.e. in the recipient; In Graph Transformation, this method is applied to attributes of host graph objects, *rightSide* being an attribute of the right side of the rule and *context* being the *match*-context built up by subsequently matching the attributes of corresponding graphical objects.

A.5 Interface Package agg.attribute.handler

This package contains the interfaces for an attribute handler.

A.5.1 Interface agg.attribute.handler.AttrHandler

public interface AttrHandler

extends Serializable

This interface is implemented by Attribute Handlers; It provides services for the Attribute Manager.

Methods

- **getName**

```
public abstract String getName()
```

Getting the name of this handler so an attribute manager can display it in the handlers' menu.

Returns: A (hopefully) meaningful name of the attribute handler.

- **newHandlerType**

```
public abstract HandlerType newHandlerType(String typeString)

 throws AttrHandlerException
```

Getting the type handle for a textual representation, for example *int* -¿ int.

Parameters: typeString - The textual representation for the type wanted.

Returns: The handle for the requested type if such exists, null otherwise.

Throws: AttrHandlerException (cf. Section A.5.6) When the type cannot be found in the handler.

- **newHandlerExpr**

```
public abstract HandlerExpr newHandlerExpr(HandlerType type,
                                           String expr)
```

```
throws AttrHandlerException
```

Getting the expression handle by providing the type and a String-representation of the expression, e.g. *3+x*.

Parameters: type - A handle of one of the types created by `newHandlerType()`.

Returns: The handle for a newly created expression or...

Throws: AttrHandlerException (cf. Section A.5.6) When the expression String cannot be a representation of an expression of the given type.

- **newHandlerValue**

```
public abstract HandlerExpr newHandlerValue(HandlerType type,
                                            Object value)
```

```
throws AttrHandlerException
```

Getting the expression handle by providing the type and an appropriate instance of the type.

Parameters: type - A handle of one of the types created by `newHandlerType()`.

Returns: The handle for a newly created expression or...

Throws: AttrHandlerException (cf. Section A.5.6) When the instance is not of the required type.

A.5.2 Interface agg.attribute.handler.HandlerType

public interface HandlerType

extends Serializable

This interface is implemented by Attribute Handlers; provides services for the Attribute Manager. It is used in the *SymbolTable*.

See Also: SymbolTable (cf. Section A.5.4)

Methods

- **toString**

```
public abstract String toString()
```

Getting the string representation of this type. Overrides the *toString()* method of the *Object* class.

Overrides: toString in class Object

- **getClazz**

```
public abstract Class getClazz()
```

Obtaining the actual class rather than just its textual representation. The name is funny because getClass() is already defined in Object as a `final` method.

Returns: A class handle.

A.5.3 Interface agg.attribute.handler.HandlerExpr

public interface HandlerExpr

extends Serializable, Cloneable

This interface is implemented by Attribute Handlers; provides services for the Attribute Manager. It is used in the *SymbolTable*.

See Also: SymbolTable (cf. Section A.5.4)

Methods

- **toString**

```
public abstract String toString()
```

Overrides: toString in class Object

- **getValue**

```
public abstract Object getValue()
```

Obtaining the value.

Returns: The value as an Object instance.

- **getCopy**

```
public abstract HandlerExpr getCopy()
```

Obtaining a copy of the message receiving expression.

Returns: The copy.

- **check**

```
public abstract void check(SymbolTable symTab)

 throws AttrHandlerException
```

Type-check the expression under a given symbol table with declarations.

Parameters: symTab - the declaration Table to use for the checking

Throws: AttrHandlerException (cf. Section A.5.6) if the checking yields an inconsitency. An exception is preferred over a return value as it is a ready-to-use propagation mechanism with specific information easily attached.

- **evaluate**

```
public abstract void evaluate(SymbolTable symTab)

 throws AttrHandlerException
```

Evaluate the expression under a given symbol table containing variable declarations and (hopefully) also the assignments.

Parameters: symTab - the declaration Table to use for the evaluation

Throws: AttrHandlerException (cf. Section A.5.6) if the evaluation yields an error (a missing value for a variable etc.)

- **isConstant**

```
public abstract boolean isConstant()
```

Checks if the expression is constant. Needed for keeping users from giving expressions that are not allowed in a context.

Returns: `true` if constant, `false` sonst.

- **isVariable**

```
public abstract boolean isVariable()
```

Checks if the expression is a single Variable. Needed for keeping users from giving expressions that are not allowed in a context.

Returns: `true` if a variable, `false` sonst.

- **isComplex**

```
public abstract boolean isComplex()
```

Checks if the expression is a complex one (like x+1). Needed for keeping users from giving expressions that are not allowed in a context.

Returns: `true` if is complex, `false` sonst.

- **equals**

```
public abstract boolean equals(HandlerExpr testObject)
```

Returns `true` if the expression equals `testObject`, `false` otherwise.

- **isUnifiableWith**

```
public abstract boolean isUnifiableWith(HandlerExpr expr,
                                        SymbolTable symTab)
```

Checks if the recipient can be *matched*, i.e. *unified* with the first parameter under a certain variable assignment.

Parameters: expr - The expression to check if unifiable with; symTab - Contains the variable assignments under which to perform the test.

Returns: `true` if the two expressions ar matching, `false` sonst.

A.5.4 Interface agg.attribute.handler.SymbolTable

public interface SymbolTable

An interface between the Attribute Manager and the Attribute Handlers. Passing of types and values of identifiers. This interface is implemented by the Attribute Manager. A Handler evaluates expressions; the values are assigned by the Manager, never by the Handler; therefore we have no assignment methods in this interface.

Methods

- **getType**

  `public abstract HandlerType getType(String name)`

  Getting the type of an identifier.

  Parameters: name - Identifier's name
  Returns: Identifier's type

- **getExpr**

  `public abstract HandlerExpr getExpr(String name)`

  Getting the value of an identifier.

  Parameters: name - Identifier's name
  Returns: Identifier's value as expression

A.5.5 Class agg.attribute.handler.AvailableHandlers

```
java.lang.Object
 |
 +----agg.attribute.handler.AvailableHandlers
```

public class AvailableHandlers

extends Object

The purpose of this class, being the only class in a package of interfaces, is that an attribute manager knows where to find its handlers. Whenever a new handler is installed, its fully qualified pathname has to be added to the static array `nameList`. That's all a new Handler has to do besides, of course, implementing those methods. All an attribute manager has to do is (besides implementing the SymbolTable interface) calling `newInstances()`. It then gets an array of attribute handler instances, one for every handler in the mentioned list.

Variables

- **nameList**

  `protected static String nameList[]`

  This is the list to extend by new Handlers.

Constructors

- **AvailableHandlers**

  `public AvailableHandlers()`

Methods

- **newInstances**

  `public static AttrHandler[] newInstances()`

  This is the method to call by an attribute manager.

  Returns: an array of attribute handler instances, one for every handler in the mentioned list.

A.5.6 Class agg.attribute.handler.AttrHandlerException

```
java.lang.Object
 |
 +----java.lang.Throwable
         |
         +----java.lang.Exception
                 |
                 +----agg.attribute.handler.AttrHandlerException
```

public class AttrHandlerException

extends Exception

Constructors

- **AttrHandlerException**

  `public AttrHandlerException(String msg)`

A.6 Interface Package agg.attribute.handler.gui

This package contains the interfaces for editors of handler expressions, type selection and handler customization.

A.6.1 Interface agg.attribute.handler.gui.HandlerEditorManager

public interface HandlerEditorManager

Returning the appropriate editors/renderers. All the return values can be null, meaning the client should use his default editors and change the edited objects by using the usual interface.

Methods

- **getCustomizingEditor**

```
public abstract HandlerCustomizingEditor getCustomizingEditor(AttrHandler handler)
```

Getting the customizing editor for an attribute handler.

- **getTypeEditor**

```
public abstract HandlerTypeEditor getTypeEditor(AttrHandler handler,
                                                HandlerType type)
```

Getting the standard type editor for a certain handler and start editing the specified type.

- **getExprEditor**

```
public abstract HandlerExprEditor getExprEditor(AttrHandler handler,
                                                HandlerType type,
                                                HandlerExpr expr)
```

Getting the instance editor for a handler type and start editing the specified expression.

A.6.2 Interface agg.attribute.handler.gui.HandlerEditor

public interface HandlerEditor

This interface is extended by handler editors, so clients can be notified of occuring changes.

Methods

- **addEditorObserver**

```
public abstract void addEditorObserver(HandlerEditorObserver obs)
```

Adds a new editor observer to the editor.

- **removeEditorObserver**

```
public abstract void removeEditorObserver(HandlerEditorObserver obs)
```

Removes an editor observer from the editor.

A.6.3 Interface agg.attribute.handler.gui.HandlerExprEditor

public interface HandlerExprEditor

extends HandlerEditor (cf. Section A.6.2)

This interface is implemented by attribute handler expression editors; it provides services for the Attribute instance (tuple) editors.

Methods

- **getRendererComponent**

```
public abstract Component getRendererComponent(HandlerType type,
                                               HandlerExpr exprToRender,
                                               Dimension availableSpace)
```

Returns a graphical component for displaying the specified expr. The `availableSpace` limit should be honoured, since this is a service for displaying the expr in a table cell. However, the renderer can contain tools (e.g. buttons) for invoking its larger custom renderer. Either `type` or `exprToRender` cannot be null.

- **getEditorComponent**

```
public abstract Component getEditorComponent(HandlerType type,
                                             HandlerExpr exprToEdit,
                                             Dimension availableSpace)
```

Returns a graphical component for editing the specified expr. The `availableSpace` is a recommendation when the editor wishes to be operatable in a compact table cell and needs not be taken into account. Either `type` or `exprToEdit` cannot be null.

- **getEditedExpr**

```
public abstract HandlerExpr getEditedExpr()
```

Returns the edited expression.

A.6.4 Interface agg.attribute.handler.gui.HandlerTypeEditor

public interface HandlerTypeEditor

extends HandlerEditor (cf. Section A.6.2)

This interface is implemented by attribute handler type editors; it provides services for the attribute type (tuple) editors.

Methods

- **getRendererComponent**

```
public abstract Component getRendererComponent(AttrHandler handler,
                                               HandlerType typeToRender,
                                               Dimension availableSpace)
```

Returns a graphical component for displaying the specified type. The `availableSpace` limit should be honoured, since this is a service for displaying the type in a table cell. However, the renderer can contain tools (e.g. buttons) for invoking its larger custom renderer. Either `handler or` typeToRender' cannot be null.

- **getEditorComponent**

```
public abstract Component getEditorComponent(AttrHandler handler,
                                             HandlerType typeToEdit,
                                             Dimension availableSpace)
```

Returns a graphical component for editing the specified type. The `availableSpace` is a recommendation when the editor wishes to be operatable in a compact table cell and needs not be taken into account. Either `handler or` typeToRender' cannot be null.

• **getEditedType**

```
public abstract HandlerType getEditedType()
```

Returns the edited type.

A.6.5 Interface agg.attribute.handler.gui.HandlerCustomizingEditor

public interface HandlerCustomizingEditor

extends HandlerEditor (cf. Section A.6.2)

This interface allows to interactively customize an attribute handler.

Methods

• **getComponent**

```
public abstract Component getComponent()
```

Returns a graphical component for customizing the handler (e.g. setting and changing options).

• **getAttrHandler**

```
public abstract AttrHandler getAttrHandler()
```

Returns the edited attribute handler.

• **setAttrHandler**

```
public abstract void setAttrHandler(AttrHandler handler)
```

Sets the edited attribute handler.

A.6.6 Interface agg.attribute.handler.gui.HandlerEditorObserver

public interface HandlerEditorObserver

This interface must be implemented by clients who wish to listen to changes in a handler editor.

Methods

- **editingStopped**

```
public abstract void editingStopped(HandlerChangeEvent e)
```

This method is invoked when the GUI user stopped editing.

- **editingCancelled**

```
public abstract void editingCancelled(HandlerChangeEvent e)
```

This method is invoked when the GUI user cancelled editing.

A.6.7 Interface agg.attribute.handler.gui.HandlerChangeEvent

public interface HandlerChangeEvent

Implementations of this interface signal changes in a handler editor.

Methods

- **getSourceEditor**

```
public abstract HandlerEditor getSourceEditor()
```

Getting the editor that signaled the change.

A.7 Java Expression Interpreter

This section contains some classes of the java expression interpreter.

A.7.1 Class agg.attribute.parser.javaExpr.ClassResolver

```
java.lang.Object
 |
 +----agg.attribute.parser.javaExpr.ClassResolver
```

public class ClassResolver

extends Object

implements Serializable

Variables

- **packages**

```
protected Vector packages
```

- **primitives**

```
protected static Hashtable primitives
```

Constructors

- **ClassResolver**

```
public ClassResolver()
```

Methods

- **init**

```
private void init()
```

- **getArrayDimensions**

```
protected int[] getArrayDimensions(String text)
```

- **getArrayClass**

```
protected Class getArrayClass(String name)
```

- **forName**

```
public Class forName(String name)
```

- **getPackages**

```
public Vector getPackages()
```

- **setPackages**

```
public void setPackages(Vector packages)
```

A.7.2 Class agg.attribute.parser.javaExpr.Jex

```
java.lang.Object
 |
 +----agg.attribute.parser.javaExpr.Jex
```

public class Jex

extends Object

implements ActionListener

Variables

- **PARSE_ERROR**

```
public static final int PARSE_ERROR
```

- **IS_CONSTANT**

```
public static final int IS_CONSTANT
```

- **IS_VARIABLE**

```
public static final int IS_VARIABLE
```

- **IS_COMPLEX**

```
public static final int IS_COMPLEX
```

- **typeTF**

```
protected TextField typeTF
```

- **parser**

```
protected static JexParser parser
```

- **out**

```
protected PrintStream out
```

- **err**

```
protected PrintStream err
```

- **redirect**

```
protected ByteArrayOutputStream redirect
```

- **redirectOut**

```
protected PrintStream redirectOut
```

- **isOutput**

```
protected boolean isOutput
```

- **refObj**

```
protected static Object refObj
```

Constructors

- **Jex**

```
public Jex()
```

Methods

- **main**

```
public static void main(String args[])
```

- **fullTest**

```
public void fullTest(String line)
```

- **actionPerformed**

```
public void actionPerformed(ActionEvent e)
```

- **getExprProperty**

```
protected int getExprProperty()
```

- **redirectToString**

```
protected void redirectToString()
```

- **restoreOutputStream**

```
protected void restoreOutputStream()
```

- **addMessage**

```
public static String addMessage(Exception ex)
```

- **parseOutputOn**

```
public void parseOutputOn()
```

- **parseOutputOff**

```
public void parseOutputOff()
```

- **parse**

```
public int parse(String text)

 throws AttrHandlerException
```

- **getPropertyText**

```
protected String getPropertyText(int code)
```

- **parse_**

```
protected int parse_(String text)
```

```
 throws ParseError
```

- **check**

```
public void check(String text,
                Class type,
                SymbolTable symtab)
```

```
 throws AttrHandlerException
```

- **check_**

```
public void check_(String text,
                 Class type,
                 SymbolTable symtab)
```

```
 throws ParseError
```

- **isAssignable**

```
protected boolean isAssignable(Class to,
                             Class from)
```

- **test_interpret**

```
protected Object test_interpret(String text,
                              Class type,
                              SymbolTable symtab)
```

```
 throws AttrHandlerException
```

- **interpret**

```
public Object interpret(String text,
                        Class type,
                        SymbolTable symtab)

 throws AttrHandlerException
```

- **interpret_**

```
public Object interpret_(String text,
                         Class type,
                         SymbolTable symtab)

 throws ParseError
```

A.7.3 Interface agg.attribute.parser.javaExpr.Node

public interface Node

Methods

- **jjtOpen**

```
public abstract void jjtOpen()
```

This method is called after the node has been made the current node. It indicates that child nodes can now be added to it.

- **jjtClose**

```
public abstract void jjtClose()
```

This method is called after all the child nodes have been added.

- **jjtSetParent**

```
public abstract void jjtSetParent(Node n)
```

This pair of methods are used to inform the node of its parent.

- **jjtGetParent**

```
public abstract Node jjtGetParent()
```

- **jjtAddChild**

```
public abstract void jjtAddChild(Node n)
```

This method tells the node to add its argument to the node's list of children.

- **jjtGetChild**

```
public abstract Node jjtGetChild(int i)
```

This method returns a child node. The children are numbered from zero, left to right.

- **jjtGetNumChildren**

```
public abstract int jjtGetNumChildren()
```

Return the number of children the node has.

- **interpret**

```
public abstract void interpret()
```

Interpret method

- **checkContext**

```
public abstract void checkContext()
```

Finding the type and checking for consistency.

- **getRootResult**

```
public abstract Object getRootResult()
```

- **dump**

```
public abstract void dump(String prefix)
```

Bibliography

[Bey91] M. Beyer. GAG: Ein graphischer Editor für algebraische Graphgrammatiksysteme. Master's thesis, 1991.

[Boo91] G. Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Bur86] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*, volume 32 of *Mathematical Research — Mathematische Forschung.* Akademie-Verlag, Berlin, 1986.

[CL95] I. Claßen and M. Löwe. Scheme evolution in object oriented models: A graph transformation approach. In *Proc. Workshop on Formal Methods at the ISCE'95, Seattle (U.S.A.)*, 1995.

[Dec94] Rina Dechter. Backtracking algorithms for constraint satisfaction problems - a survey. Technical report, University of California, Irvine, September 1994. `ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/backtracking.ps`.

[Ehr79] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73*, pages 1–69, 1979.

[Fla96] D. Flanagan. *Java in a Nutshell. A Desktop Quick Reference for Java Programmers.* O'Reilly and Associates, Inc, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[HMTW95] R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In G. Valiente and F. Rossello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science.* Technical Report B - 19, Universitat de les Illes Balears, 1995.

[Koc97] M. Koch. Bedingte verteilte Graphtransformation und ihre Anwendung auf verteilte Transaktionen. Technical Report 97-11, 1997.

[LB93] M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, '93, LNCS 690*, pages 451–456, 1993.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[Mel97] B. Melamed. Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem. Studienarbeit, 1997.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Rat97] Rational software cooperation. Unified modeling language. available via *http://www.rational.com*, 1997.

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

[Rud96] M. Rudolf. Konzeption und Implementierung grundlegender Aspekte der Ableitungskomponente eines Graphtransformationssystems. Studienarbeit, 1996.

[Rud97] Michael Rudolf. Konzeption und implementierung eines interpreters für attributierte graphtransformation. Master's thesis, 1997.

[Sch92] G. Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme.* Arbeitsberichte des Institus für mathematische Maschinen und Datenverarbeitung (Informatik), University of Erlangen, 1992.

[Sch93] H.-J. Schneider. On categorical graph grammars integrating structural transformation and operations on labels. *TCS*, 109:257 – 274, 1993.

[Sch97] A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[Sun97a] Sun Microsystems, http://www.suntest.com/JavaCC/. *Java Compiler Compiler*, 1997.

[Sun97b] Sun Microsystems, http://java.sun.com:80/products/jdk/. *The Java Development Kit (JDK)*, 1997.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.

[TFKV98] G. Taentzer, I. Fischer, M Koch, and V. Volle. Visual design of distributed systems by graph transformation. to appear as article in graph grammar handbook 3 "Concurrency and Distribution", 1998.

[Wag97] A. Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras*. PhD thesis, TU Berlin, 1997.

[Win95] B. Windolph. Vom Grapheditor AGG zu einem Graph-Grammatik-System. Master's thesis, 1995.

[Wol97] D. Wolz. Colimit Computations for Graph Structures and Algebraic Specification Languages. PhD Thesis, TU Berlin, FB Informatik, in preparation, 1997.

[Zün95] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, RWTH Aachen, 1995.