vis-A-vis: An Object-Oriented Application Framework for Graphical Design-Tools

Horst Lichter, Kurt Schneider

University of Stuttgart, Software Engineering Group, Breitwiesenstr. 20/22, W-7000 Stuttgart,
lichter@informatik.uni-stuttgart.de

Abstract

Many engineering disciplines use graphical notations with defined semantic meaning, such as Petri nets, or structure charts. The diagrams drawn in these notations represent semantic models upon which application-specific operations may be performed. To handle this type of notations and their semantic models comfortably, a graphical editor can support them. As there are many different graphical notations in every discipline, reuse of general editor functionality should be supported and encouraged to save tool building effort.

vis-A-vis is an object-oriented framework serving this purpose. However, instead of laying the burden of properly composing those classes on the tool builder´s shoulders, vis-A-vis also contains a backbone architecture. In this paper we motivate the construction of vis-A-vis by listing the essential requirements that we had set out for a framework supporting in tool building. The scope of vis-A-vis based tools is defined, and a short survey of the main concepts of vis-A-vis is given. An example of a System Dynamics graphical notation editor which was built within vis-A-vis illustrates these concepts and shows how little effort a tool builder has to spend to implement an editor.

Keycodes: D.1.5, D.2.2, H.5.2
Keywords: Object-oriented Programming, Tools and Techniques, User Interfaces

1. Introduction and Motivation

Graphical notations are used in every engineering discipline, when expressive models will be constructed. Tools are needed to create such models in a graphical notation and to modify them interactively. These tools have to meet the following requirements:

- They have to provide rich *graphical capabilities*, and should be *easy to use*.
- They have to "understand" the *semantic* of the diagram which means that the tools build an internal semantic model of the designed diagram.

New high resolution monitors, powerful workstations, and window-systems allow to use interactive graphical tools. Nevertheless, it is hard to implement graphical design-tools from scratch which meet these requirements. As mentioned in Bischofberger (1990), an *application framework* consits of an object-oriented class library together with a default-application which can be enhanced and modified. The default application and the class library implement everything which is needed for every application belonging to the framework´s domain. Hence, an *application framework* supports and fastens the development of applications belonging to its domain.

Our paper explains the main concepts and the design of the object-oriented application framework *vis-A-vis*. The framework is especially designed to fasten the construction of interactive design-tools for graphical notations. It provides a uniform architecture, a uniform look & feel for all tools, and a set of graphical representations which may be used in constructing a special design-tool.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

In section 2, we define the application domain of the framework. Section 3 summarizes some requirements which have influenced the design of vis-A-vis. In section 4, we give a short overview of the application framework. Section 5 illustrates, by an example, how a new tool is constructed within vis-A-vis. Section 6 describes our experiences, the state of the development, and what remains to be done.

Throughout this paper we use the following terminology: with the notion "vis-A-vis tool" we mean a design-tool built within the application framework. The application framework itself is called "vis-A-vis".

2. The Application Domain of vis-A-vis

vis-A-vis is an application framework which is especially designed to build editors for graphical notations. vis-A-vis provides mechanisms and reusable classes that allow to develop a special editor very quickly and in a uniform way. Graphical notations are used to construct and explain *models*. These models are often called "plans" or "designs".

vis-A-vis tools allow to draw models in a graphical notation consisting of symbols for objects and relations, and to operate on the constructed internal model. In every engineering disciplines, graphical notations exist which fall into the application class of the framework. In our field, Software Engineering, we find graphical notations e.g. for Structured Analysis, Data Flow Diagrams, Jackson Structured Programing, Coad/Yourdon Object-Oriented Analysis, SADT, State-Transition-Diagrams, or Booch Object-Oriented Design. In other disciplines vis-A-vis might be used to build editors for e.g. organigrams, activity networks, or circuit diagrams.

vis-A-vis provides mechanisms to perform either operations on a single semantic object of the model or on the entire model. Simulations, consistency checks, or conversions are often performed on the entire model.

The application framework vis-A-vis is tailored to fasten the development of the above-mentioned domain of design-tools. This results in the following advantages:

- The framework is small, it has about 60 classes which may be directly reused by a tool builder.
- The framework offers mechanisms used in every editor, so only a minimum effort of programming is needed to produce a new editor. A new editor can be built according to a method outlined in section 5.

One the other side, vis-A-vis is not useful to build tools which do not lie in the domain of applications described above. This is the main difference between vis-A-vis and more general application frameworks like ET++, which is explained in Gamma (1989). Furthermore, vis-A-vis is not designed to build arbitrary user interfaces of arbitrary applications. vis-A-vis is no UIMS (User Interface Management System). The look and feel of vis-A-vis tools is predefined by the framework, and should not be modified by a tool builder.

An Example of a Graphical Notation with Semantic

The elements of every graphical notation are symbols which have a well defined semantic in most notations. The symbols can be divided into two groups:

- Symbols which represent objects of the model.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

- Symbols which represents relations between these objects.

Assume the well-known Petri-net notation as presented e.g. in Baumgarten (1990). The elements of Petri-nets are conditions, transitions, and connections between conditions and transitions and vice-versa. A transition fires, if all input conditions of the transition are marked. Conditions are represented in the graphical notation by a circle. If a condition is marked, a small gray circle is shown in its center. A transition is visualized by a short horizontal line; the connections are represented by solid arrows.

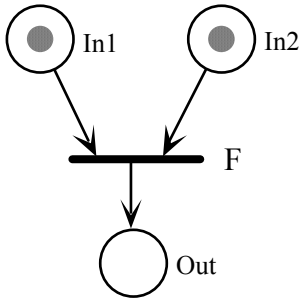


Fig. 1: A Petri-net

In our example, these symbols have the following semantic: a transition is associated with a function and a condition is interpreted either as input or output data.

The Petri-net shown in figure 1 models a function “F”, which has two inputs, “In1” and “In2”, and produces the output “Out”.

3. Requirements for vis-A-vis

When designing vis-A-vis, we had three perspectives in mind: What does the user of a vis-A-vis tool expect, what can be done for easy tool building, and what are general software engineering requirements that should be met?

3.1 Requirements of a Tool User

If someone uses a graphical design-tool, he presumably expects to find:

- An easy-to-use tool handled similar to one of the well-known drawing tools, and equipped with serveral basic operations, such as moving, deleting, aligning of symbols, storing and loading of models.
- Uniform look & feel: Learning how to use one vis-A-vis tool helps using all of them. This is particular important for all graphical operations, but holds for the mechanisms of invoking semantic actions, too.
- Problem specific symbols and connections: With one mouse-click the entire symbol should be placed or moved; no symbols need to be manually reconstructed from basic graphics. The offered symbols, labels, and lines correspond in appearance and style to those in textbooks.
- Screen size does not limit the size of models. The vis-A-vis screen window is just a view on a virtually much bigger drawing region. There must be mechanisms to navigate in this bigger space.
- Unlike traditional drawing-programs, vis-A-vis tools should be aware of a model´s structure: Labels and connections are recognized as belonging to a basic symbol and are moved together with it.
- Semantic awareness allows graphical syntax constraints checked by the tool: All possibilities of a notation should be available, but nothing more. If the notation does not use dotted line, a line should not possibly be dotted.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

- The semantic models should be as independent of vis-A-vis as possible, to be easily reused elsewhere, e.g. one model can be viewed under different aspects in different tools, without the need to rebuild it.

3.2 Tool Builders´ Requirements

When going about implementing a tool to support a graphical notation, one hopes to:

- Build a problem specific tool as fast as possible, with as little effort as possible. To achieve this, a big library of reusable building blocks should be available.
- Clear and small interfaces should allow building a vis-A-vis tool without needing too much knowledge of vis-A-vis itself. As a side effect, using these building blocks and interfaces should lead to a standard architecture of every new tool.
- Each of the offered building blocks must be adaptable; particularly, there must be a way to mask graphical features of a prefabricated building block, if the notation does not allow to use them.
- Problem-specific symbols, semantic, help texts and arrangements of labels should be easily integrated, without worrying about every detail of how to connect them with the rest of the tool.
- Easy access to the file system and to a database for persistent storing of the semantic models should be available without much coding.

3.3 Requirements from the Perspective of Software Engineering

To enforce software qualities like maintainability and extendability, vis-A-vis should provide:

- Strict separation of the graphical notation and its corresponding semantic model to allow independent development and change.
- Highly generic code with little redundancy to minimize change anomalies.
- Recognition of general tool quality criteria as introduced in Schneider (1991).

3.4 Our Conclusion from the Requirements

We decided to design and implement vis-A-vis in an object-oriented way. Thus, a new vis-A-vis tool is built with the application framework in setting parameters and more often in inheriting features from general vis-A-vis classes. We chose Smalltalk-80, because this programming language and its class library are well suited to build graphical user interfaces.

Conforming to the concept of object-oriented application frameworks, vis-A-vis offers a set of classes as building blocks for direct use or as abstract superclasses. This is superior to the traditional library approach, since lower documentation effort is needed and higher maintainability is obtained.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

4. An Overview of vis-A-vis

In this section, we will explain some features of vis-A-vis. First, we show, how the user interface of vis-A-vis tools looks like and we introduce the visualizations which vis-A-vis offers to build graphical notations. Furthermore, we present the standard architecture which the framework defines for every vis-A-vis tool.

4.1 The Look & Feel of vis-A-vis

Figure 2 shows the standard vis-A-vis user interface.

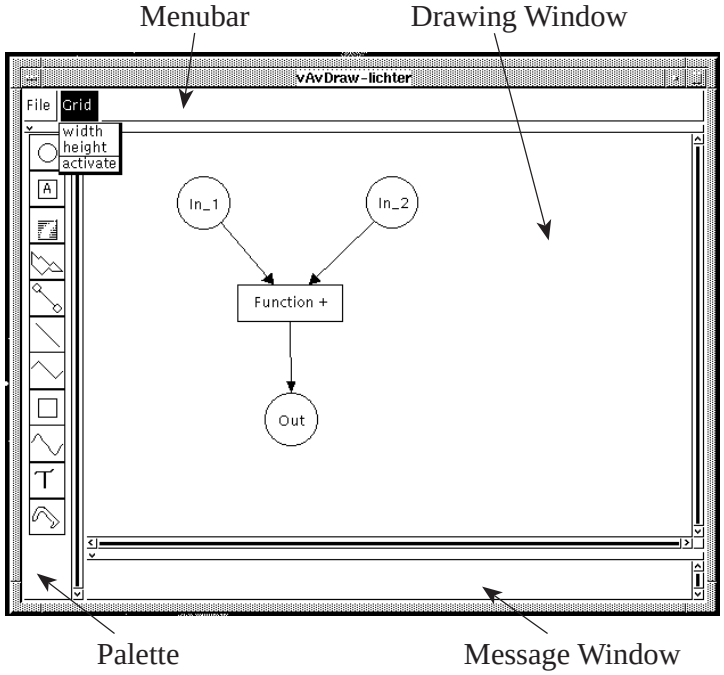


Fig. 2: The user interface of vis-A-vis tools

The user interface has four regions:

The *Palette* shows icons which represent the symbol types (the elements) of a graphical notation. After the user has selected a symbol type in the palette, he can place a corresponding symbol in the drawing window.

The *Drawing Window* is used to build the model in the graphical notation. The window can be scrolled horizontally and vertically. The symbols can be moved and selected to obtain its semantic menu.

The *Menubar* offers tool-specific menus. vis-A-vis provides two menus by default: the *File* menu offers functions like “save”, “load”, “print”, or “quit”, the *Grid* menu allows to align the symbols shown in the drawing window.

The *Message Window* is used to inform the user of errors or to display online help information. vis-A-vis provides a mechanism to show help information for every icon of the palette.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

Kinds of Visualizations

vis-A-vis offers a set of visualizations in its class library which may be used to build a graphical notation for a vis-A-vis tool. Figure 3 shows the kinds of visualizations which vis-A-vis provides at its current state of development.

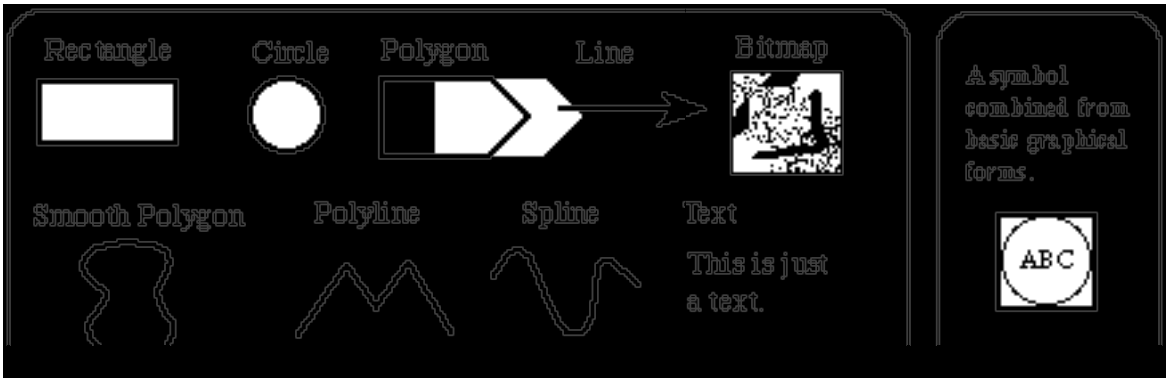


Fig. 3: Kinds of visualizations offered by vis-A-vis

These visualizations have specific graphical attributes. Rectangles, circles etc. may have e.g. an inside color; lines may be drawn in different styles, such as dotted or terminated by an arrow; texts may be drawn in different fonts and styles. These basic visualizations may be combined by a tool builder to design a complex symbol. Figure 3 shows a symbol which is the result of the combination of a rectangle, a circle, and a text.

4.2 The Architecture of vis-A-vis Tools

vis-A-vis defines a standard architecture for every tool constructed within the framework. This architecture is mainly influenced by the following aspects:

- The "Tool-Material-Metaphor" introduced in Budde (1992) describes, how object-oriented interactive tools should be designed.
- The strict distinction between the symbols of a graphical notation and their associated semantic.
- The construction of user interfaces according to the Model-View-Controller-Paradigm, described in Krasner (1988).

The architecture can be illustrated by levels of abstractions. It has three levels: the *User-Interface-Level*, the *Tool-Level*, and the *Material-Level*. The following figure shows these levels and their main components.

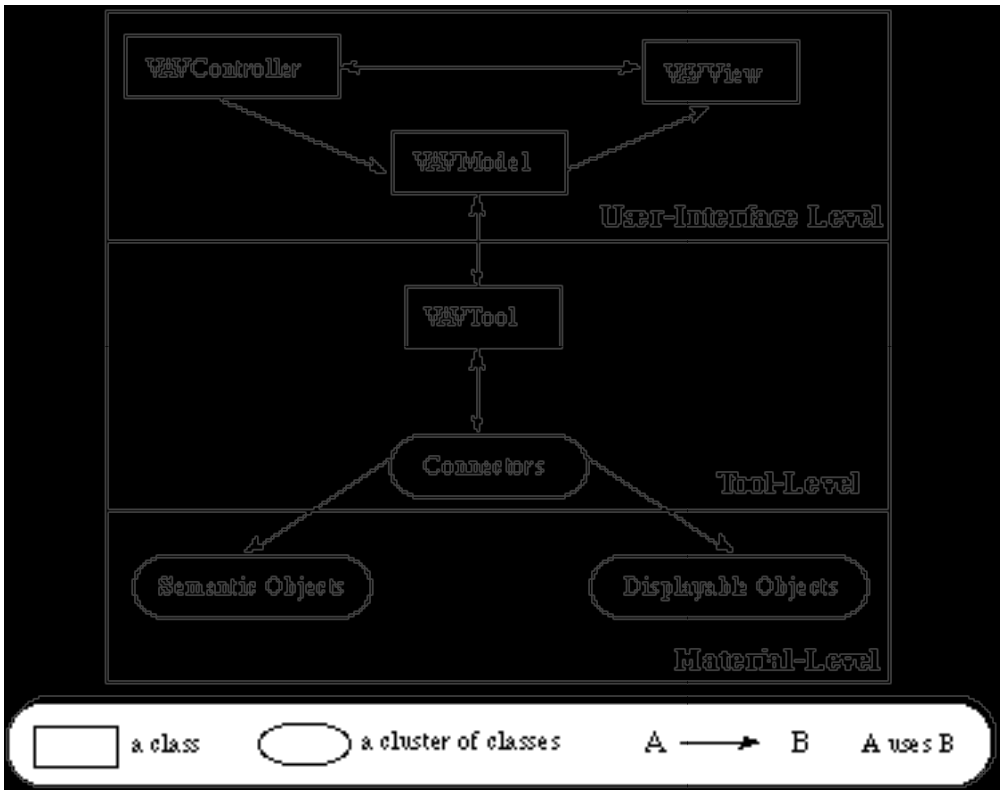


Fig. 4: The architecture of vis-A-vis tools

The User-Interface-Level

This level contains all classes necessary for the look and feel of the tool. These classes were designed according to the Model-View-Controller-Paradigm. The controller class is responsible for the interaction between the user and the tool. The view class determines, how the tool looks like, and the model class stores the state of user interaction and connects the interface classes to the tool.

The Tool-Level

The tool is divided into several components. These are:

- A main component, implemented in the class *VAVTool*, which is responsible for the central mechanisms and for the general functions like “save”, “load”, etc. Furthermore, it is adapted to the requirements of the vis-A-vis user interface classes and integrates the other components explained below. The class *VAVTool* is always used as superclass, when a new tool is implemented with additional functionality.
- Components called *connectors* link the semantic object associated with a symbol to its graphical representation, and propagate the operations which can be performed on the semantic object.

Connector classes must be implemented for every symbol type presented in the palette of a tool. For this purpose, vis-A-vis offers a set of predefined connector classes which may be directly reused when constructing a new tool.

The Material-Level

This level contains the semantic classes, implemented for every symbol type of the graphical notation, and the corresponding visualization classes. As mentioned before, vis-A-vis offers a set of predefined visualization classes which can be used to design the symbols shown in the tool.

4.3 Main Concepts and Mechanisms of vis-A-vis

First, we explain in more detail the concept of connectors which is necessary to understand how vis-A-vis tools are structured.

The Concept of Connectors

vis-A-vis clearly distinguishes between the semantic objects which are the elements of the internal model, and their graphical representation. For this purpose, we have developed some basic mechanisms and a message protocol to link the semantic object to its representation in a very loose coupling. These mechanisms are implemented in a tree of classes called *connector classes*. The following figure illustrates the relation between a vis-A-vis tool, its connectors, their semantic and representation objects.

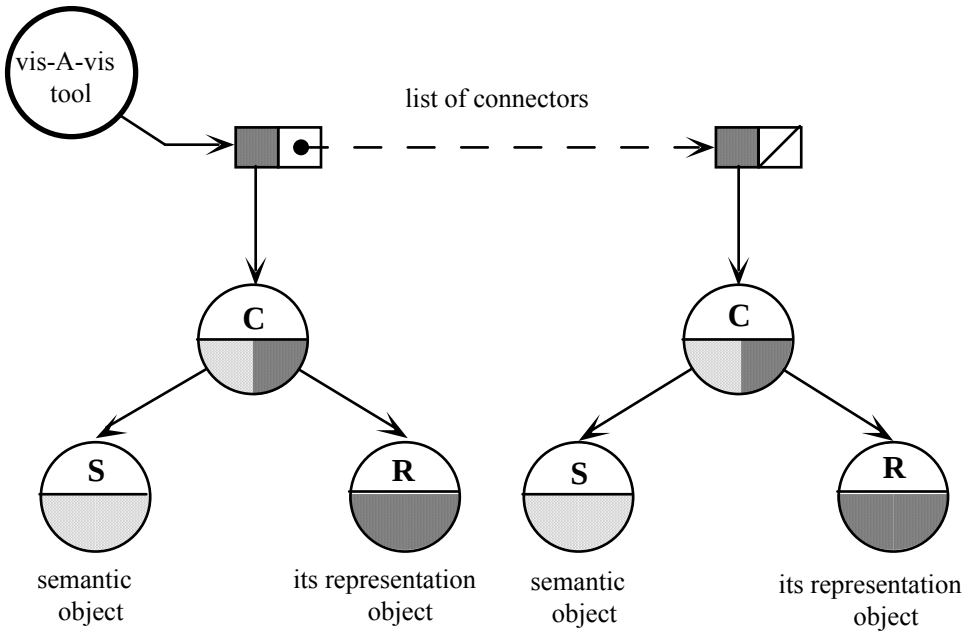


Fig. 5: The internal structure of a vis-A-vis tool

A vis-A-vis tool stores a connector for every semantic object of the model in an internal list. Every connector links, by two references, its semantic objects to the corresponding visualization. A connector takes control, when the visualization of its semantic object is selected in the drawing window of the tool. The connector is responsible to offer the menu of its semantic object to the user, and forces the semantic object to perform the associated operation, when a menu item is selected.

A connector defines, how its semantic object should be visualized and how a change of the semantic object should modify its visualization. vis-A-vis allows that a connector may have

any number of *internal connectors*, since a visualization is often assembled of different graphical forms, as shown e.g. in figure 3. Every internal connector is used to visualize a specific aspect of the semantic object. The usage of internal connectors can be illustrated by the visualization of the semantic object "condition" in the notation of Petri-nets which is visualized by a circle. Two aspects of conditions should also be represented in the visualization:

- A condition may have an identifier which should be displayed on the right side of the circle.
- If a condition is marked, a little gray circle should be added in the center of the condition symbol.

A connector which implements this type of visualization needs to use two internal connectors: One which links the identifier of the condition to a displayable text, and a connector which links the marked-attribute to a displayable circle. The following figure shows the structure of a *ConditionToCircleConnector* with its internal connectors.

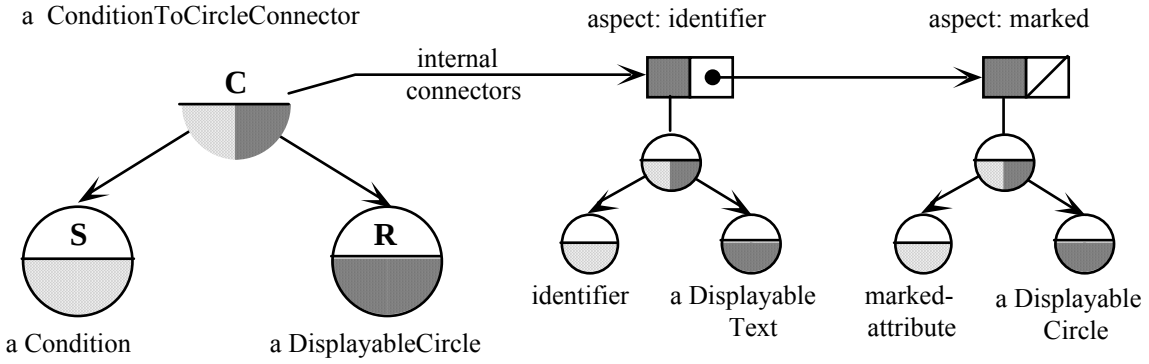


Fig. 6: The usage of internal connectors

The described concept of linking a semantic object to its representation guarantees that the semantic classes can be designed without thinking about the visualization in a tool. Furthermore, it is possible to modify or replace the visualization of a semantic class by another one, without performing any change in the implementation of the semantic class.

4.4 The vis-A-vis Class Library

The vis-A-vis class library consists of about 60 classes implemented in Smalltalk-80. This library contains two important sublibraries (beside more marginal others): The *connector library* and the *displayables library*. The displayables form a class tree consisting of classes like *DisplayableRectangle*, *DisplayablePolygon*, or *DisplayableLine*. The implementation of the connector classes is based on the displayable classes. In the following, we give a short overview of the connector library, since these classes are reused when a new tool is constructed.

The Connector Library

vis-A-vis offers a set of abstract connector classes which are subclasses of the class *XtoYConnector*, where "X" means any semantic class and "Y" stands for any visualization class. The structure of this library is shown in figure 7. Class *XtoYConnector* implements all mechanisms which every connector has to know, like linking a semantic object to its

visualization, or the interaction with its internal connectors. On this highly abstract and generic level neither “X” nor “Y” is defined, this is deffered to their subclasses.

The connector class liberary offers a set of abstract connector classes which are adapted to a special visualization. In a first classification step, connectors distinguish, whether their displayable object is two-dimensional (fills an area), one-dimensional (non-area-filling line), or if it is a text. According to this first step of replacing “Y” by concrete graphical objects, three groups of (still rather abstract) connector classes are provided:

- Connector classes defining the visualization as an arbitrary two-dimensional geometric object. For this purpose the class *XtoAreaConnector* is implemented which enhances the general functionality of class *XtoYConnector*. Direct Subclasses of *XtoAreaConnector* are *XtoCircleConnector*, *XtoImageConnector*, *XtoPolygonConnector*, and *XtoRectangle–Connector*. *XtoCircleConnector*, for example, is an abstract class determining that an arbitrary semantic object is linked to the visualization of a displayable circle. The class *ConditionToCircleConnector*, implemented for a Petri-net editor, should be a subclass of *XtoCircleConnector*.
- Connector classes defining that the visualization is an arbitrary one-dimensional geometric object. The root of this subhierachy is *XtoPolyLineConnector* which has the subclasses *XtoLineConnector* and *XtoSplineLineConnector*.
- Connector classes defining that the visualization is a text. Two connector classes are included in the library: The class *XtoTextConnector* implements the general functionality to visualize an arbitray semantic object as a text and the frequently used class *StringToText–Connector* which defines that the semantic object displayed is an object of class String.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

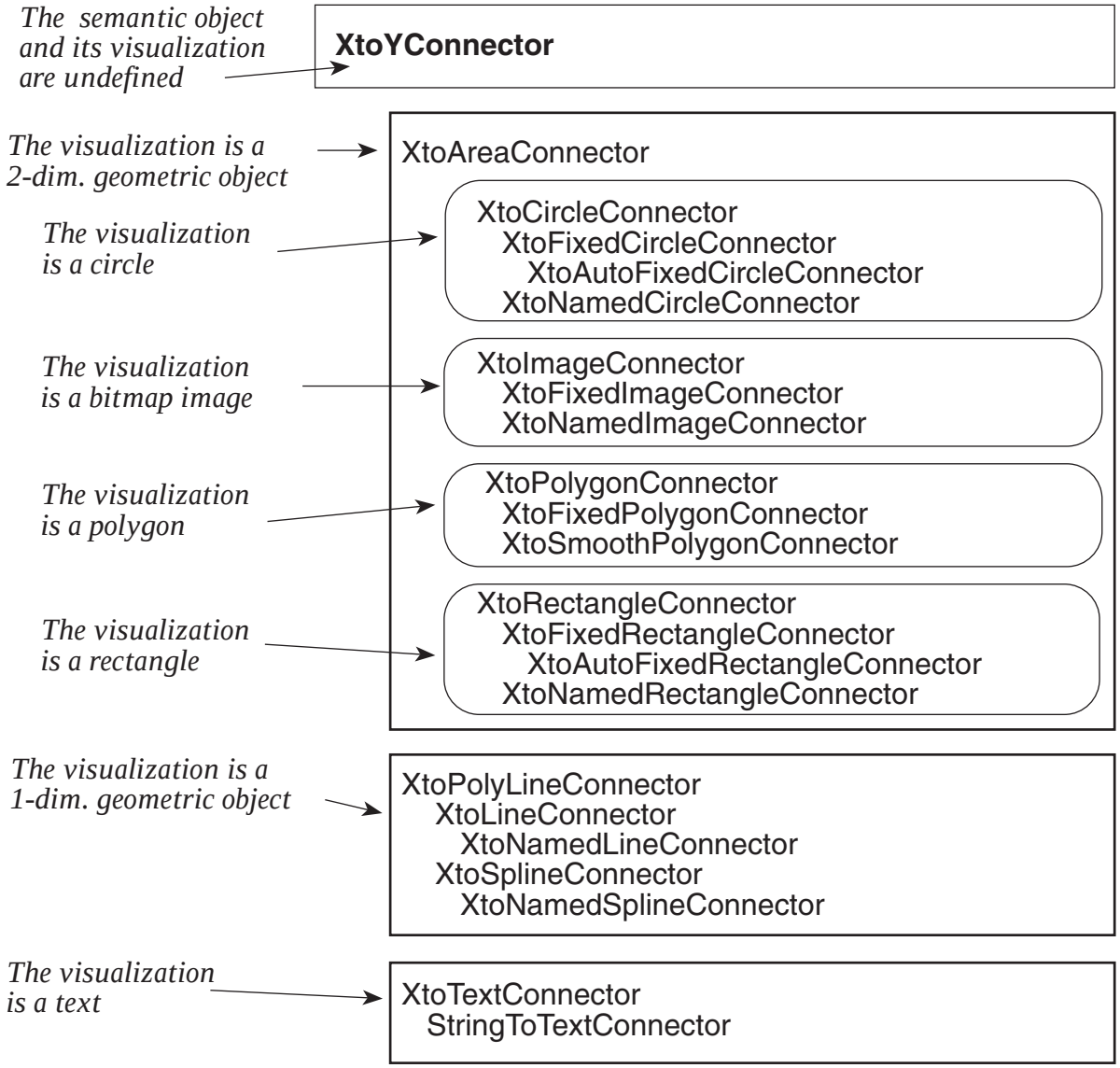


Fig. 7: The connector class library

Each of these three classes is once again specialized for the following purposes:

- Since in many graphical notations symbols are combined of a one or two-dimensional geometric object and a text, subclasses are provided which allow to display both the geometric object and the name. These classes are called *XtoNamed<Y>Connector*, where <Y> is a concrete geometric object, like a rectangle or a circle (e.g. *XtoNamedRectangle–Connector*). The text itself can be manipulated via an internal connector which must be a sublass of *XtoTextConnector*.
- The second group of subclasses is provided to construct connectors in which the shape of the geometric object can not be manipulated by the user of a vis-A-vis tool. These classes are called *XtoFixed<Y>Connector*, e.g. the class *XtoFixedPoylgonConnector* implements that the visualization is a specific polygon always displayed according to its definition (concerning its shape, inside color, line width and so on). The definition is given by the tool builder.
- Even more specialized subclasses implement a two-dimensional geometric object together with a name (text); they allow to automatically adapt the size of the geometric object to

the length and width of the text displayed in its center. These classes are called *XtoAutoFixed<Y>Connector*. *XtoAutoFixedRectangleConnector*, for example, can be used to construct a connector which links an arbitrary semantic object to the visualization of a rectangle which adapts its shape to the text displayed in its center.

While this paragraph discussed specialization of "Y" towards concrete graphical objects, nothing has been said about specializing "X", too. This is up to the tool builder to determine the semantic associated (connected) to the concrete substitute of "Y".

5. How to Build a New Tool within vis-A-vis - An Example

In this section, we give an example of a simple vis-A-vis tool with semantic. This tool supports System Dynamics, a well-known simulation approach using a graphical notation. We show what a tool builder has to do in detail when constructing a new tool, and what is added by vis-A-vis.

5.1 The Application: System Dynamics

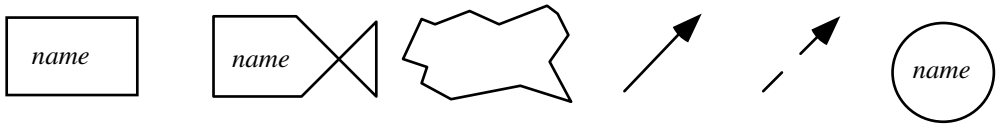
The history of System Dynamics dates back to the 1960´s. Forrester (1961) first introduced it as "Industrial Dynamics". Its initial purpose was to model, simulate and predict the complex dynamic behavior of a company, and how its evolution could be affected by managerial decisions and policies. The formalism was then applied to the extended field of simulating the entire US economical development (umemployment, average income etc.). The approach was then called "System Dynamics", as Forrester observed its general quality: It could be applied to model *any* dynamic system.

System Dynamics models a system in the mathematical form of a set of difference equations. Simulation time proceeds in fixed parts of Δt. System Dynamics assumes a Δt short enough to allow these "jumps of time" without significant distortion of model behavior. Thus, a simulation is just a sequence of transformations of the initial system state. $State_{n+1}$ is calculated as function of $state_n$ and Δt. The state of a system is defined by the values of so-called *levels*. These values represent the essential objects and their quantities within the model. One step in simulation time simultaneously calculates and transforms the values of all levels at $time_n$ to their subsequent values at $time_{n+1}$.

A graphical notation allows to sketch the structure of a System Dynamics model, indicating which levels are increased or decreased by (possibly other) levels. Feedback loops appear frequently in System Dynamics diagrams when levels influence each other. Traditionally, the diagrams had to be transformed manually to a textual language, called DYNAMO. Some details had to be added, as diagrams show only a model´s structure, but not all its details necessary for simulation. Diagrams, however, bridge the communication gap between experts of application and simulation.

Only recently, graphical tools like STELLA appeared on the market. We intended to implement such a graphical Systems Dynamic tool within the vis-A-vis framework.

The symbols of System Dynamics graphical notation are:



level rate sink / source material flow dependency auxiliary variable

The semantic of symbols:

- *Levels* hold values which together make up the state of the system. Levels store their values and update them after each simulation step. The values represent quantities of the material under study.
- *Rates* increase or decrease level values, i.e. they introduce the dynamic aspect into the models. They control the flow of material. Material may flow from one level to another, or one of the partners may be an anonymous source or sink.
- *Sources* or *sinks* do not have a name or behavior. They just represent "anywhere outside the system" and have conceptually unlimited capacity.
- To show the way and direction of *material flow*, levels and sinks/sources are connected by arrows with solid lines.
- A rate may depend on nay levels to calculate the flow of material it controls. Dotted arrows show such *dependencies*.
- If formulas within rates grow too complex, *auxiliary variables* may be introduced to split up the calculations for better understanding. Auxiliary variables do not store values (like levels do), they only help rates to perform their calculations. Therefore, auxiliary variables appear only between dotted arrows.

Figure 8 is a screen hardcopy and shows how our editor tool looks like. We added a problem specific *Simulation* menu to the *File* and *Grid* popup-menues inherited by every vis-A-vis tool. The popped up *Simulation* menu is shown: The user may reset the simulation clock, set parameters of the simulation, and start it. If the graphical model design is not complete (i.e. it violates a structure constraint) or if one of its parts is not fully specified, an error message appears. In figure 8, a simple predator-prey-model is sketched. No auxiliary variables were needed, as we used rather trivial dependency functions: As well prey as predator are increasing in number by birth, and decreasing by death. Besides that, prey provides additional food and better living conditions for predators, leading to more fertility on their side. Predators eat prey. Decreasing their number, they reduce their own additional food resource: A feedback loop on the diagram.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

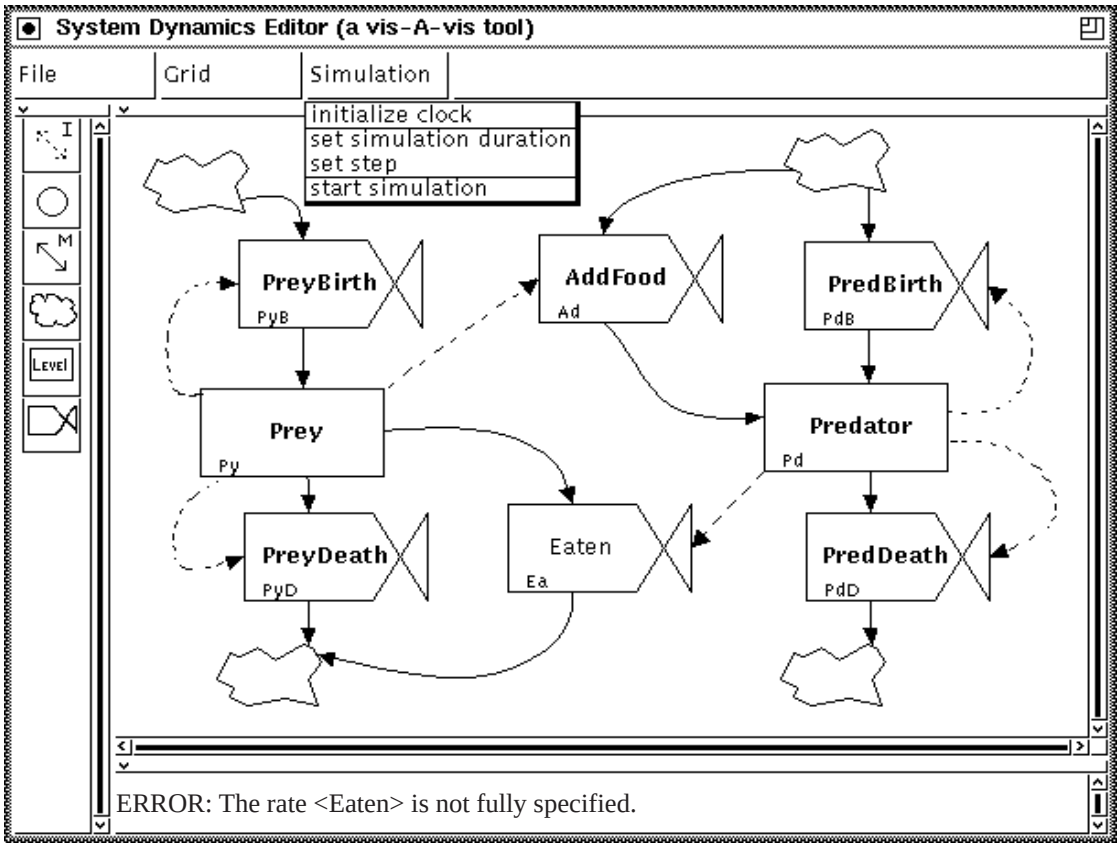


Fig. 8: The System Dynamics editor

We choose to use bold style for the names of fully specified objects, and plain style for levels lacking an initial value, or rates without a formula assigned (such as *Eaten* in figure 8). To specify hidden features of a symbol´s semantic object, the user simply clicks on it and gets a menu of available operations. In our example he might set, delete or change names, initial values or formulas, respectively. This will lead to an immediate adjustment of the name style. In the lower left corner of each level and rate symbol, an acronym of the name is displayed. The System Dynamics editor derives them from the symbol names and checks them to be unique. Acronyms may be used to write down formulas in a more compact form. In our example they illustrate, how two features of one semantic object (name and acronym) may be visualized within one symbol. The mechanism of internal connectors is used to easily place them and modify their styles with only marginal programming effort.

5.2 How we built the System Dynamics Editor

We suggest the following method to develop a tool within vis-A-vis:

1. *Identify the semantic objects of the application and implement them.*

   Our rate class, for example, has name, acronym and formula attributes. It may evaluate its formula and may perform some other simulation-related duties. The semantic objects also have to provide the semantic menu which appears upon clicking into their representations, and the semantic operations triggered by this menu.

2. *Determine basic representations (symbols) for the semantic objects.*

   For the System Dynamics editor, we could directly reuse the classes *DisplayableCircle* and *DisplayableRectangle* from the vis-A-vis library of displayables. To draw sinks/sources and rates, we reused the class *DisplayableFixedPolygon* and parameterized it with the specific shapes of the symbols. Each of the symbols also needs an icon to be placed in the palette. Icons can be drawn with a bitmap editor included in vis-A-vis. Furthermore, a help text (explaining the symbol´s usage to the user) may be provided in the tool´s configuration file.

3. *Choose which aspects of the semantic objects are to be visualized, and how that should look like.*

   We had to write connector classes for each semantic class, such as *RateToPolygon-Connector* which connects the semantic objects (rates) with their visualization, a *DisplayablePolygon*. As a subclass of *XtoFixedPolygonConnector* of the vis-A-vis connector library, *RateToPolygonConnector* inherits all important vis-A-vis features like menu access and message propagation, graphical modification operations for the polygon, and full integration into the other vis-A-vis services.

4. *Finally, build a subclass of the vis-A-vis class VAVTool. This subclass contains the tool features performed on the entire model.*

   In our example, all *Simulation* menu items trigger operations of a System Dynamics simulator frame which is linked to the editor. Frequently, application programs will be first implemented without a graphical notation interface. We had a command-driven simulation frame completed and wanted to tie it up to a graphical editor. Before using the graphical editor, the net had to be woven textually, in a DYNAMO-style command-driven fashion: A tedious job!

5.3 Effort Spent for Editor Construction

When implementing the graphical System Dynamics editor we did not have to care for symbol drawing or moving operations. vis-A-vis´s *DisplayableSplineLines* could be used to draw arrows with smooth edges. All arrows follow when a symbol is moved. Without any additional effort, semantic objects are connected and notified if their representations are connected by arrows. Only icons and help texts must be provided; their arrangement in the palette and the message region is handled by the installation routine. Semantic objects and the editor must provide a semantic menu description (list of items) and the corresponding operations. To form a working menu from this descriptions and propagate a service request back to the tool or to the responsible semantic object, is up to vis-A-vis. Saving and restoring on files needs no additional effort.

A student implemented the System Dynamics editor (not including the simulation frame) within 40 person-hours. Since this editor was one of the first applications we realized within vis-A-vis we expect to implement comparable tools in significantly shorter time, as experience grows and documentation matures.

6. Conclusion: Current and Future Work

Today, vis-A-vis

- offers a big variety of graphical options and building blocks;
- successfully connects semantic models to diagrams;

- only weakly supports general mechanisms for semantic models;
- stores models in the object-oriented database GemStone which, however, needs writing a schema class for each semantic class to be stored;
- does not support a tool builder by meta-tools which could help him to implement a new tool with still less effort;
- consists of some 60 classes corresponding to about 10,000 lines of Smalltalk code.

Our work has now reached a phase of consolidation and inspection. Some applications (an editor for software architecture, sketched in Lichter (1992), and SESAM simulation editors described in Ludewig et al. (1992)) were started, which revealed some deficits and stimulated improvements. Right now, integrating revolutionary new features is defered until our implemented concepts will be evaluated and improved.

Hand in hand with evolution and improvement of the framework itself, we also extend and improve all aspects of documentation: Besides the existing architecture and code documentation, we work on a general user guide for vis-A-vis tools and a tool builder´s handbook (with examples).

Future work will be devoted to better support for semantic model building, a new model-transaction and version-control-concept and a few meta-tools to further automate tool building!

References

Baumgarten, B. (1990): Petri-Netze Grundlagen und Anwendungen. Bibliographisches Institut, Mannheim.

Bischofberger, W. (1990): Prototyping-Oriented Incremental Software Development - Paradigms, Methods, Tools and Implications, Doctorial Thesis, Johannes Kepler Universität Linz.

Budde, R., M.-L. Christ-Neumann, K.-H. Sylla (1992): Tools and Materials, an Analysis and Design Metaphor. G.Heeg, B. Magnussen, B. Meyer (eds.), Proceedings of the Seventh International Conference TOOLS Europe ´92, Prentice Hall.

Forrester, J. W. (1961): Industrial Dynamics, M.I.T.-Press, Cambridge, MA.

Gamma, E., A. Weinand, R. Marty (1989): Design and Implementation of Et++, a Seamless Object-Oriented Application Framework. Structured Programming, Vo. 10, No 2.

Krasner, G., S. Pope (1988): A Cookbook for using the Model-View-Controller User Interface in Smalltalk-80, Parc Place Systems.

Lichter, H. (1992): Architecture Prototyping - A Seamless Transition into the Final System. Proc. of the NordDATA´92 conference, Tampere, Finland.

Ludewig, J, Bassler, Th., Deininger, M., Schneider, K., Schwille, J. (1992): SESAM - Simulating Software Projects, Proc. of the SEKE-92 conference, Capri, Italy; IEEE Comp. Soc. Press.

Schneider, K. (1991): Systematische Evaluierung von CASE-Tools, Proc. of the TOOL´91 conference, Karlsruhe, Germany, vde-Verlag, Berlin.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.

Trademarks

STELLA is a trademark of High Performance Systems, Inc.
GemStone is a trademark of SERVIO.
Smalltalk-80 is a trademark of Parc Place Systems, Inc.

In Proceedings of the IFIP Tc5/Wg5.10 Working Conference on interfaces in industrial Systems For Production Engineering (March 15 – 17, 1993). J. Rix and E. G. Schlechtendahl, Eds. IFIP Transactions, vol. B-10. North-Holland Publishing Co., Amsterdam, The Netherlands, 43-57.