JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik (Systemsoftware)

Documenting Object Interaction Using Behaviour Diagrams

Stefan Chiettini

Report 13
April 1999

Author's address:

Institut für Praktische Informatik (Systemsoftware)
Johannes Kepler Universität Linz
A - 4040 Linz / Auhof

e-mail:
stefan.chiettini@ssw.uni-linz.ac.at

JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik (Systemsoftware)

Documenting Object Interaction Using Behaviour Diagrams

Stefan Chiettini

Report 13
April 1999

Abstract

The documentation of object-oriented systems usually consists of two parts: First there is the static part with the description of the classes and methods. This part holds all information about interfaces, inheritance relations, and aggregations. The second part, which is the topic of this paper, describes the dynamic behaviour of the system in a certain situation at run time. Common design and documentation techniques like OMT [Rum91] or UML [Fow97] introduce event trace diagrams (OMT) and sequence diagrams (UML) to visualize run time behaviour of interacting objects. These diagrams show the message sequence in a certain situation at run time. Their major weakness is that they are themselves static and therefore capable to illustrate only one special case, not general behaviour.
This paper proposes *behaviour diagrams* as an extension of existing diagrams to meet the requirements of modern documentation: structured documents, interactive exploration of documentation, and multimedia capabilities. A prototype system is introduced that implements these extensions.

Contents

1. Introduction

Modern object-oriented systems focus on reuse (code reuse in class libraries, design reuse in frameworks, see Figure 1). In most cases the reusable classes are part of a huge code base that has to be understood in order to locate and use them correctly. The size and complexity of these code bases has increased to a level where it is impossible to understand their structure without appropriate documentation.

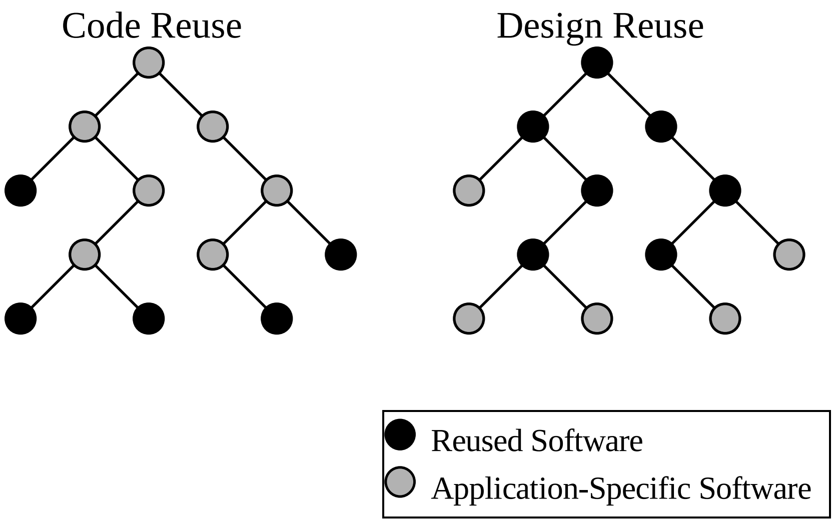


Figure 1: Code Reuse versus Design Reuse

The reusable parts must be customized and/or parameterized to use them correctly, in many cases it is necessary to know not only the static structure but also the object collaboration at run time. When adapting a framework, for example, the programmer supplies only chunks of code (i.e. extensions) that are plugged into the framework and get called at certain times. For the correctness of the overall system, it is crucial to understand under which circumstances the extensions are called and what is the state of the system at those moments. It might also be necessary to know in which order extensions are invoked and which classes and methods are involved at all in a particular scenario. Since in most systems the reused parts are numerous and complex this is a hard task.

To help the user understand the context of the system, good documentation is therefore an indispensable part of any codebase providing reusable software. Modern documentation should meet many requirements. It should be detailed enough for maintenance programmers yet universal for new system users. It should be easy to build, easy to maintain and easy to use.

2. Structuring Documentation

The documentation of object-oriented systems should not be presented in one large block at the deepest level of detail. Since there are many potential users of the system, documentation should be structured according to the required abstraction level for each user group [Fre96], [Meu97]. To guarantee appropriate information for all user groups, it is necessary to split the documents and to provide different entry points. In order to reach all information, the documents are linked together using hypertext links. Different entry points should be provided in the form of different start documents. Splitting the documents keeps irrelevant information away from particular users.

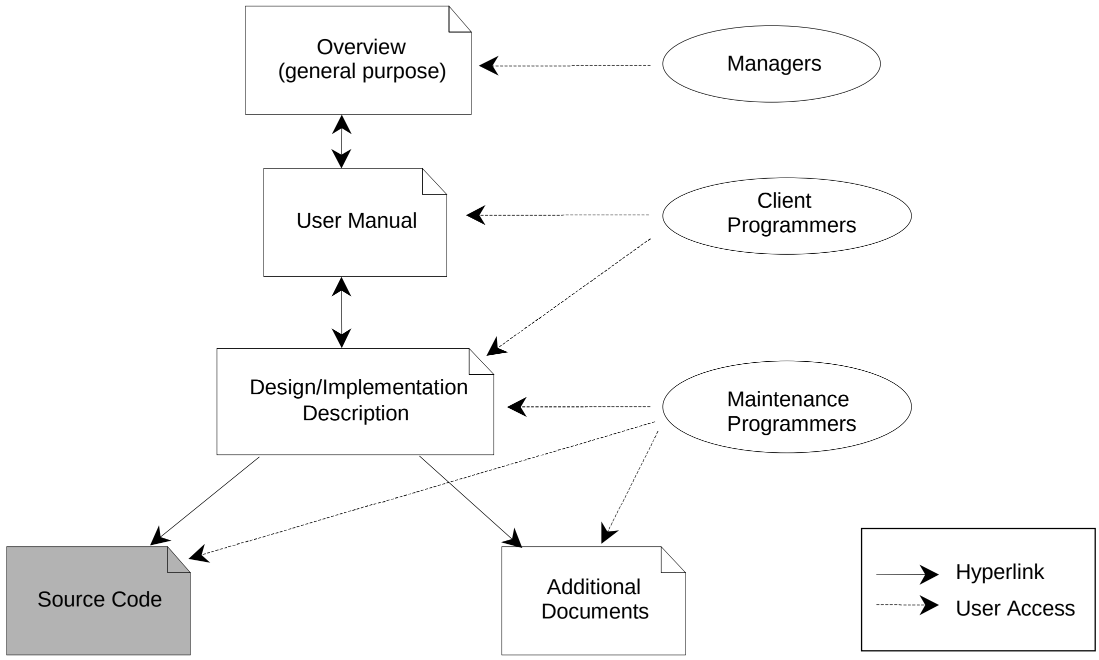


Figure 2: Documentation Structure for Different User Groups

The structure shown in Figure 2 meets the requirements of the different user groups:

- Project managers and executive managers who have to decide whether a system shall be used for a particular project or not. These users need an accurate yet short description of the purpose and the functionality of the system. They will start their investigation of the documentation at the point where the overall purpose of the system is described. In several cases they may be interested in how to use the system, too, but will rarely need detailed information about the internal structure.

- Client Programmers new to the system need a description on how to use it in order to produce working applications in a short time. Advanced users will also need direct access to detailed documentation of the system context (static and dynamic documentation).

- Maintenance programmers need detailed documentation, too, but also require access to the source code (the most finegrained level of information). Links to the source code normally are a special form of hyperlinks. They are discussed in chapters 4.4. and 5.7. Eventually existing additional documents (design specifications, results of analysis phases or standard documents like RFCs or specifications of standards) should be integrated into the documentation for fast access.

3. Documentation Techniques and Tools - An Overview

This section gives an overview of existing documentation techniques and tools. They are grouped by their focus:

- documentation of static structure ('Static Documentation')
- documentation of run time behaviour ('Dynamic Documentation')

3.1. Static Documentation

The following information is part of static documentation:

- class descriptions (purpose, general description, inheritance relations)
- method descriptions (interface, parameters, return values)
- attributes
- relations and cooperations between classes

This sort of documentation is essential for the reuse of classes. A vast number of tools exists to support the generation of static documentation. UML editors may be used to document inheritance relations between classes. Descriptions of classes, methods, and attributes are supported by hypertext authoring tools. Pattern languages [Joh92] are useful when structuring documents. Using HTML enables online availability of software-documentation on the internet.

Some commercial tools on this wide field are

- javadoc (www.javasoft.com)
- Rational Rose™ (www.rational.com)
- SNiFF+ (www.takefive.co.at)
- ObjectManual™ (www.obsoft.com)
- CDOC Professional™ (www.swbs.com)
- AutoDocs™ (www.covesoft.com/mitchell/autodocs.html)
- ...

These tools analyse source code and extract meta information about inheritance relations, aggregations, type information and embedded comments. They generate class trees, hyperlinked indexes, and keyword databases. Sometimes links to the appropriate position in the source code are generated.

In addition to these commercial tools a series of experimental systems have been developed:

- *DIF* [Gar90] and *SODOS* [Hor86] use a database system to collect documents and associate them with each phase of the software life cycle. The database is used to maintain and query documents.

- *HyperCASE* [Cyb92] combines the concepts of hypertext and CASE tools. It provides a user interface with a hypertext editor and various browsing possibilities. It also contains a repository and a data dictionary.

- *LaSSIE* [Dev90] uses a knowledge base to react on user questions and provides several views of a software system (code, architecture, feature, ..).

- *SLEUTH* [Fre96] focuses on different user groups like software engineers reusing the system, project managers, or maintenance programmers. Hypertext is used to provide flexible navigation, a query mechanism allows keyword search.

All these tools rely on the static structure of the software system, many parse source code to automatically create documentation. Hypertext has proven its usability in these tools and is a quasi-standard for good documentation now, though it is often extended with a separate query mechanism for keyword search. Documents are structured and indexed, in frameworks sometimes cookbooks [Kra88, Pre95] are provided to explain how certain adaptations have to be done.

3.2. Dynamic Documentation

The first step in learning how a system works is to know the classes, their capabilities and limitations. The next step is to understand how the objects interact at run-time. Especially frameworks rely heavily on complex object interaction, including calls to abstract classes where hotspots [Pre95] are located.

In many cases, an extension of the framework requires detailed knowledge about the framework context. In these situations, it is essential to know the involved classes, their state at a given time, and their behaviour.

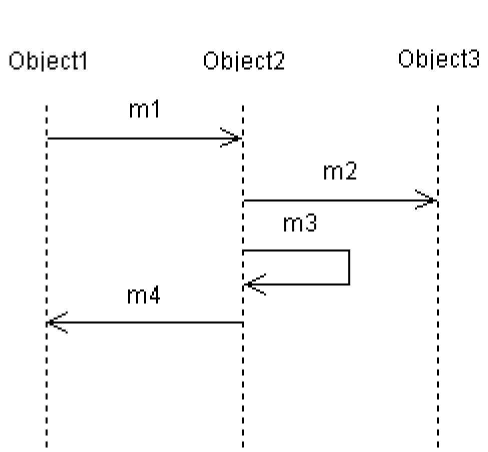


Figure 3: Event Trace Diagram in OMT

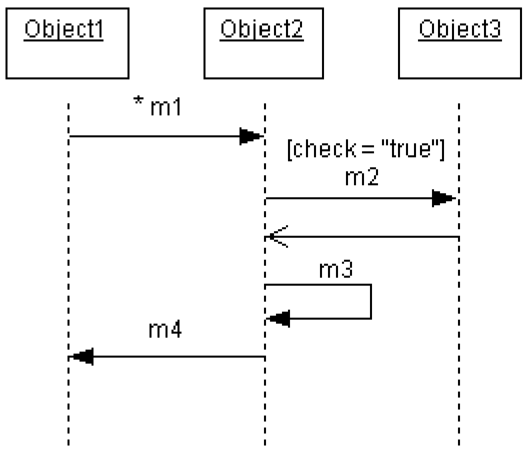


Figure 4: Sequence Diagram in UML

To provide appropriate information on object interaction, *event trace diagrams* (Figure 3) where introduced in the OMT-Method [Rum91]. They display objects as vertical lines representing the timeline. Messages sent between objects are horizontal arrows. Event trace diagrams are frequently used in object-oriented literature to illustrate object interaction (e.g. [Gam95], [Jac94]). The major weakness of these diagrams is, that no information about the control flow is shown. To fix this problem, messages in UML *sequence diagrams* [Fow97] may contain *conditions* and *iteration markers* (Figure 4). Some UML tools (like Rational Rose^{TM}) use bars instead of dashed lines to represent an object.

The diagram in Figure 4 shows that message `m1` is sent many times and that message `m2` is sent only if condition `[check = "true"]` holds. This information about the control flow is not present in Figure 3.

An extension of event trace diagrams are *scenario diagrams* introduced by [KM95]. Scenario diagrams are generated by running instrumented code. They can be inspected with a tool named *Scene* (see Figure 5) with several browsing capabilities such as collapsing parts of the diagram and linking additional documentation into the diagram. They rely heavily on hypertext and hypermedia.

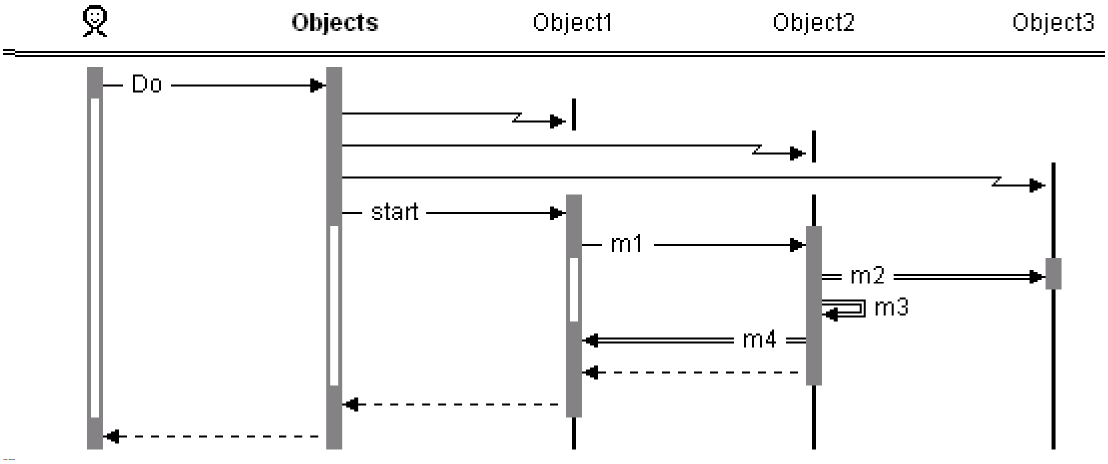


Figure 5: Scenario Diagram in *Scene*

What is missing in sequence diagrams as well as in scenario diagrams, is information about the actions that take place in the method bodies. A possible question may be: what actions (other than message sends) are performed between the receipt of message `m1` in `Object2` and the sending of message `m2` to `Object3`?

Another difficulty is to show dynamically bound method calls. Let's take a look at the following example:

```
Shape shp;
...
shp.draw();
...
```

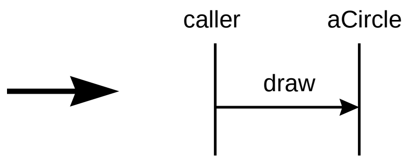


Figure 6: A Simple Scenario

This may be part of some sort of graphics editor. `Shape` may be the superclass for several concrete shapes like circles, lines, or rectangles. An event trace diagram only shows one special case (called *scenario* in OMT and *use case* in UML), for examle the case when `shp` is of dynamic type `Circle` (Figure 6). The information that `shp` could also be some other subtype of `Shape` is lost.

Even with the UML extensions added, documentation of the control flow is still too coarse grained. Iteration markers do not show how many iterations are performed. Additionally it is almost impossible to show the following situations:

Repetition of message *sequences*:

```
while ... {
  obj1.m1();
  obj2.m2();
  obj3.m3();
}
```

Conditional message *sequences*:

```
if ... {
  obj1.m1();
  obj2.m2();
}else
  obj3.m3();
```

The only possibility in a sequence diagram is to separately place the same iteration marker or condition on all affected messages. This obscures the fact that the messages form a sequential block. `else` - branches need to be marked with the inverse condition of the `if` - statement.

The conclusion is that information about general behaviour like conditional messages and dynamically bound method calls cannot be packed into a diagram which is itself static and is therefore able to illustrate only one particular scenario. To switch between different receiver objects when documenting a dynamically bound method or a conditional message, it is necessary to use a diagram which is embedded into an interactive environment, which allows exploration of several independent paths through the flow of control.

4. Behaviour Diagrams - An Extension of Sequence Diagrams

We propose some extensions to existing sequence diagrams which solve most of the problems described above. Behaviour Diagrams are extended sequence diagrams and provide the following capabilities:

- documentation of method bodies and dynamically bound method calls
- interactive use of the diagram, different paths through the diagram
- hypertext capabilities and links to the source code
- multimedia capabilities for advanced documentation features (pictures, videos, ...)

4.1. Behaviour Diagrams

One of the problems stated in Chapter 3.2 is that we don't see what happens inside the method bodies. The objects are just represented by vertical bars (Scene) or dashed lines (OMT, UML). Our basic idea is therefore to widen the bar to a textbox to show the essential control flow inside. The contents of the box shows pseudocode at an appropriate abstraction level. At the one extreme it could be just a sequence of message calls, at the other extreme it could be almost as detailed as the source code. Figure 7 shows the general layout of a behaviour diagram compared to an equivalent sequence diagram.

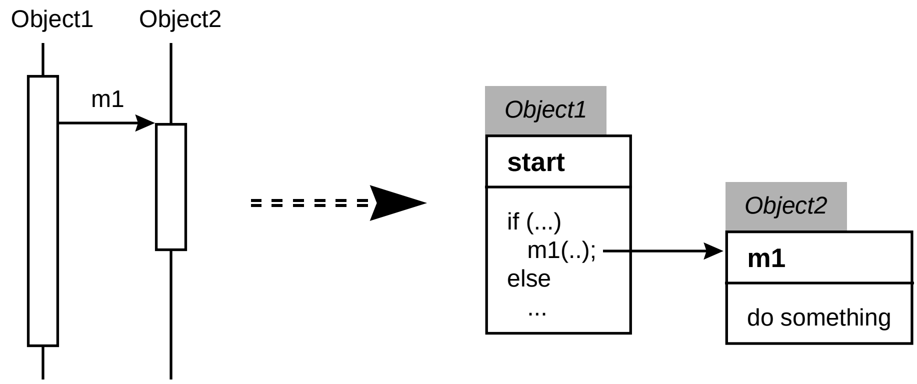


Figure 7: Sequence Diagrams versus Behaviour Diagrams

To get a maximum level of usability and flexibility, hypertext with embedded text elements (e.g. links to other boxes) is used to display the information in a box. The text elements are similar to those used in the Oberon System [MK95]. The hypertext is highly extensible - newly implemented text elements can be inserted at run time. All text is completely free for editing - no special format or syntax is required by the system.

The most important fact cannot be illustrated using printed paper - behaviour diagrams are interactive. Not all boxes are visible from the beginning. Method `start` contains a *LinkElem* that can be clicked with the mouse and pops up the referenced box (in Figure 7 this is `Object2`). A click on the link to an already visible box hides the box. This enables the user to explore parts of the diagram and hide paths that are no longer interesting.

4.2. Behaviour Diagrams - An Example

We will introduce our diagrams using a well known example - the design pattern 'Composite' [Gam95]. The static structure looks like this:

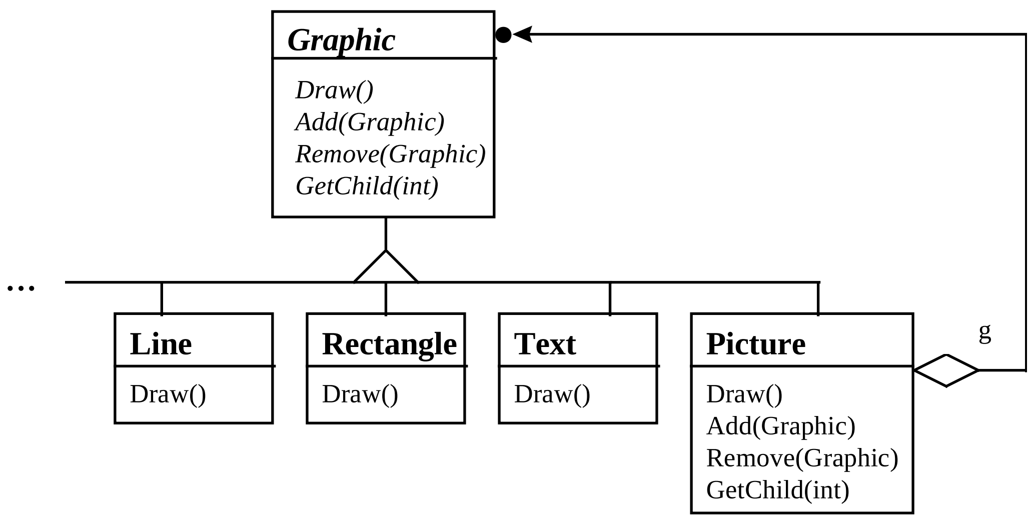


Figure 8: Possible Structure of a Simple Graphics Editor

The subclasses `Line`, `Rectangle`, and `Text` define primitive graphical objects. The `Picture` class defines an aggregate of `Graphic` objects. `Picture` implements `Draw` in such a way that it calls `Draw` on all of its children.

In Java the method body of a client (possibly a graphic editor) may look like this:

```
...
for(int i=0; i<objects.size(); i++){
     Graphic graph = (Graphic)objects.elementAt(i);
     graph.Draw();
}
...
```

Sending the `Draw` message invokes the `Draw` method of one of the concrete objects. Which particular object is addressed, depends on the configuration of the system at run time.

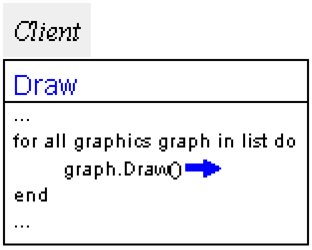


Figure 9: Initial Situation

Figure 9 illustrates the initial situation in a typical behaviour diagram. Only one starting object is visible (in this example the client object of the graphics editor). This object represents the root for all further investigation of the diagram. Multiple roots and different root types (method calls or user interaction) are possible. Clicking on the *LinkElem* (the arrow right of the word `Draw()`) pops up the referenced object `graph` (see Figure 10).

Figure 10 shows the scenario when a single line is drawn. The additional borders around object `graph` indicate that the `Line`-object is not the only scenario referenced by the link. In fact the `Line`-object is only the top of a set of objects implementing the interface of class `Graphic`.

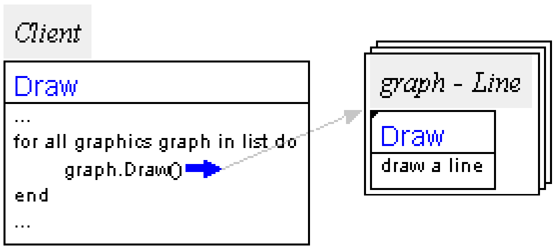


Figure 10: Possible Scenario: A Line is drawn.

Alternative objects like `Picture` or `Rectangle` are hidden behind the top object and can be switched to the foreground (see Figure 11).

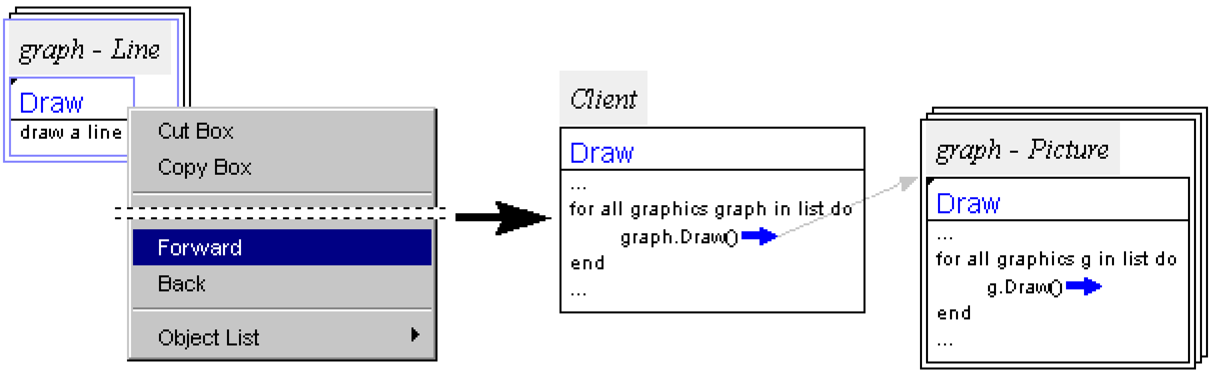


Figure 11: Switching an object to the foreground

The `Picture` object contains a *LinkElem* which represents a reference to another `Graphic` object. This link may refer to a `Line`, a `Rectangle` or again a `Picture` object.

A recursive reference via Attribute `g` in the `Picture` object is shown by a special link called 'self reference' (see Figure 12). Such a reference should not be confused with a programmatic recursion, a recursive reference can be seen as general behaviour attribute of a `g` object (like the back link from `Picture` to `Graphic` in Figure 8).

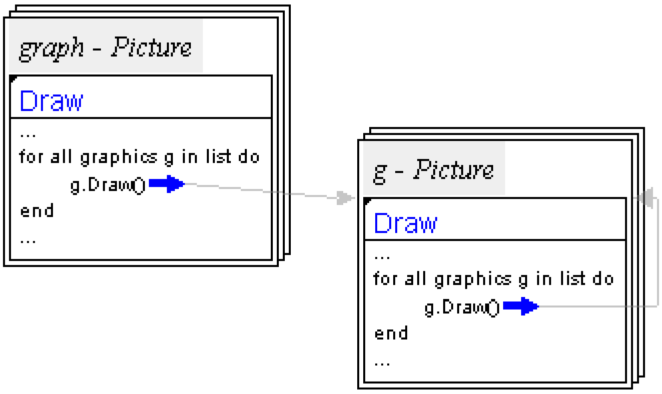


Figure 12: Recursive Reference into a *Picture* object

The method `Draw` is a *common method*, it is in the interface of the superclass and must be present in all subclasses. A special mark in the upper left corner of a method indicates that it is a common method.

4.3. Browsing in Behaviour Diagrams

Initially only the first object in the calling chain or the initiating user input is visible. Given this starting point, the user follows the links of his interest and doing so builds up a path through the diagram. The timeline which was fixed in sequence diagrams is replaced by a directed graph, which can be traversed by the user in the order of the control flow. Links that are not interesting for a particular user can be omitted. The irrelevant information stays invisible and therefore does not consume valuable space on the screen.

Lack of space on the screen is one of the main problems when using graphical notations. The fact that behaviour diagrams show additional information in the method bodies makes it difficult to place many boxes on the screen at the same time. To reduce consumtion of space, box positions are automatically chosen by the system, placing them near related boxes and avoiding overlapping boxes or arrows. Methods can be displayed in a collapsed state, where only the method header is visible (Figure 13).

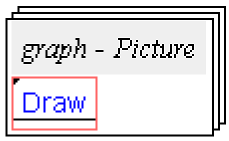


Figure 13: Collapsed *graph*

Considering these measures and especially the fact that there is only a small part of the complete diagram visible at the same time, behaviour diagrams remain small and easy to survey, even if they consist of lots of boxes. One should also take into account that the user focuses on a single method most of the time. When following a link, the user's focus moves to the new method (similar to hypertext browsing where most of the time the actual document is in the center of the user's attention).

4.4. Linking Diagrams to Source Code

It is essential to include links to the original source code of the system being documented. No matter how detailed hand written documentation is, the best (and never outdated) source for information is the code that is actually run. Many developers refer to the source code as the "only reliable documentation, that is never outdated". To gather information about details, studying source code is the best solution (also see Chapter 2).

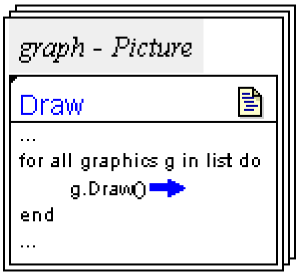


Figure 14: *SourceLinkElem*

In behaviour diagrams source code links are provided by *SourceLinkElems*, represented by the icon (see Figure 14). Source links can be placed in the text freely, the target position in the source file is free, too.

The source linking mechanism was designed to work with existing development systems, this means that the source files are not changed. The link itself consists of a URL which holds the name of the file and a fingerprint which denotes the location inside the file. To keep up consistency it is essential to link to the original source code which is actually compiled and run (and typically not resides on the same machine as the diagram). The source code is usually stored on some dedicated server and is maintained by commercial development environments. It should not be moved or copied. Copying source code is dangerous because changes in the original source are not noticed by the documentation system, besides copying source code consumes disk space. The only practicable solution is to access source code at the location where it is initially stored.

The files are located using the FTP protocol, requiring the installation of an FTP server on the machine hosting the sources. Since not all users should have access to the source code, password protection of the FTP directory is necessary. The fingerprint in the source file is robust against changes one line before or after the marked position. Source links are not generated, the target position in the source text is selected by hand. Navigation in the source file is supported by a keyword search mechanism and an integrated Java parser to allow direct jumps to method headers.

5. Implementation

The behaviour diagram editor is part of a larger software documentation system called **S.I.D.** (Structured Interactive Documentation). S.I.D. integrates a hypertext system for static documentation and behaviour diagrams for dynamic documentation. Hypertext and behaviour diagrams are interlinkable.

5.1. Attributes of S.I.D.

The tool had to meet the following requirements:

- extensibility at run-time
- easy-to-use browser
- easy-to-use editor
- independence from target language and target system
- possibility of distributed documentation

Extensibility at run time is achieved by using a registry that manages the available text elements (such as link elements). This enables the user to implement new elements eventually needed for special purposes. New elements can be added to the application together with a menu item and a hotkey at run time (see Chapter 5.3.).

One main goal was to keep the tool as simple as possible. Since writing documentation is an unpopular task, production of documentation should not be hard and complicated work. Text inside the boxes is edited inplace. Standard formatting is done by hotkeys and the toolbar. To draw a diagram no complicated dialogs are needed. Actually most of the time not even the mouse is needed to build a diagram.

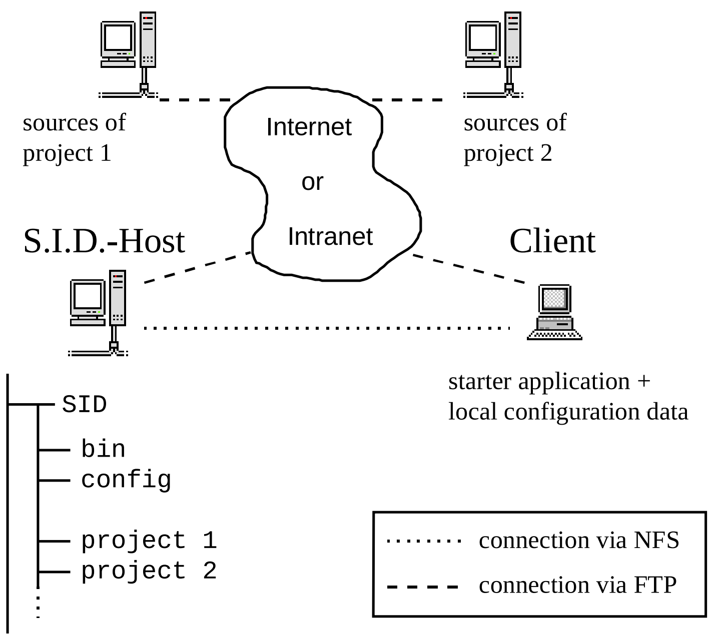


Figure 15: Structure of S.I.D.

To integrate the tool into existing distributed development systems, S.I.D. is capable to run in a network as well as on a local computer. Due to the implementation language Java, S.I.D. is independent from the target system.

S.I.D. maintains its projects in a directory located on a server which is accessed via NFS. Each S.I.D. project is encapsulated in its own subdirectory and contains the complete documentation of exactly one object-oriented system (framework, class library, ..). The client uses a small starter application to load local configuration files and start the application.

5.2. Browsing Distributed Documentation

All documentation files generated with S.I.D. can be browsed online via the internet. An applet serves as starting point for interactive exploration of behaviour diagrams and hypertext documentation. Documents contain links in the URL format which makes it even possible to follow links to other servers or take a look at the source code (if access is permitted).

The Applet is a class named *SIDApplet*, it may be opened by using a HTML file that looks like this:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" content="text/html;
charset=iso-8859-1">
<TITLE>S.I.D. Internet Demonstration</TITLE>
</HEAD>
<BODY BGCOLOR="#c0c0c0">
<applet name=SIDApplet
     code="SIDApplet.class" width=400 height=150
     archive="signedSID.jar"
     archive="windows.jar"
     archive="swing.jar"
     codebase="http://pclab14.ssw.uni-linz.ac.at/">
     <param name=servername value="pclab14.ssw.uni-linz.ac.at">
</applet>
</BODY>
</HTML>
```

This Applet connects to a Server (class *SIDServer*) that knows about the documentation location and provides the requested documents. The Server returns a list of all available documentation projects and provides access to the appropriate index file of a requested documentation tree.

5.3. Installing and Running S.I.D.

S.I.D. is delivered as a JAR-Archive (SID.jar). This archive has to be expanded into a directory which is part of the system path and the CLASSPATH.

The following packages are needed to run S.I.D.:

JDK version 1.1.6 or later
Swing version 1.0.3 or later

For MediaElems to work, the Java Media Framework 1.0 (better 1.1) has to be installed.

The application is started by the command '`java SID.SIDStarter`', which opens the main window (see Figure 16). This window serves as starting point for every S.I.D. session. It is also the central point to open or create projects and to edit the configuration of a given project. Additionally it manages the window list of all open windows of a particular project.

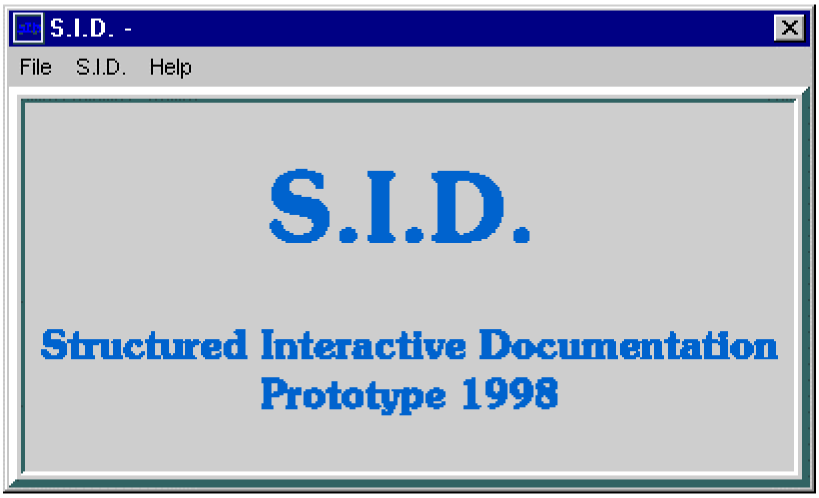


Figure 16: S.I.D. Main Window

In the File menu of this window new projects can be created (Figure 17) or existing ones can be opened by choosing from a list (Figure 18). S.I.D. is language independent, but the target language can be choosen when creating a new project. This enables support for source code browsing using a built-in parser.

Figure 17: *Create Project* Dialog

Figure 18: *Open Project* Dialog

When a new project is created, the directory structure and the default index file are generated by the system. The index file serves as starting point for the creation and investigation of the document hierarchy. The documents (as well as the index) are hypertexts (created by *`File/New Text`*) which can be edited by the user (font – size, style, color, ...see Figure 19).

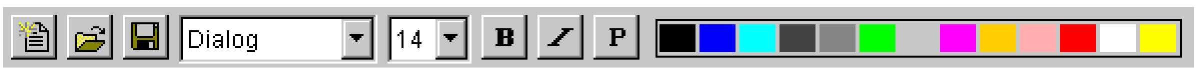


Figure 19: Toolbar with Editorfunctions

A hypertext document may host special elements called ‚text elements‘. These elements float in the text just like normal characters, they can be inserted, copied or deleted. Text elements are extensible, new ones can be added to the system at run time using a text element registry (see Figure 20). In this registry elements are inserted and provided with a menu name and a hotkey.

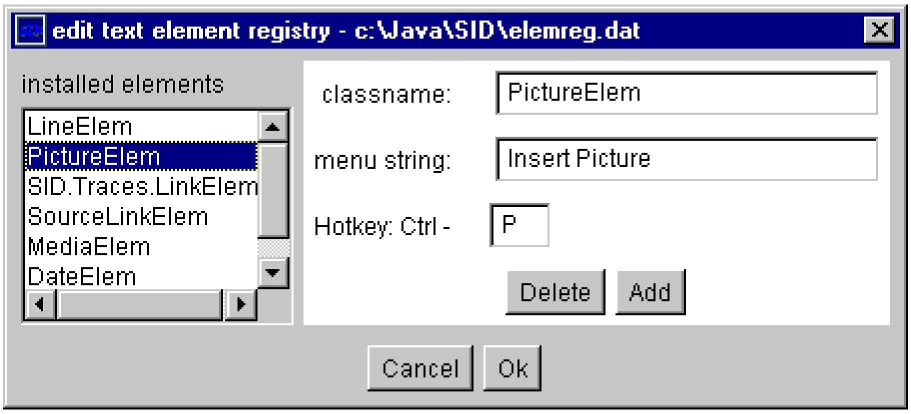


Figure 20: Registry for Textelements

The registry uses this information to build a menu labeled '`Elements`', which holds the specific Menu entries and hotkeys for the insertion of registered text elements.

Predefined Elements:

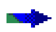
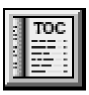
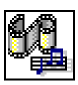
14-Oct-98	DateElem	When saving the hosting text, this element stores the date of the save operation. This enables some simple version control of documents.
	MarkElem	This is the target element for 0 - n hyperlinks. This enables hyperlinks to point somewhere into a document, not only at the beginning.
	HyperLink	The anchor of a hyperlink. This element points to 0 or 1 MarkElem. It may also point at the beginning of a document (in this case no MarkElem is necessary).
	SourceLink	This link points to a certain location in an ASCII sourcefile. This location is represented by a fingerprint.
	LineElem	A dividing line with variable thickness that can be changed manually or automatically.
http://www.ssw.uni-linz.ac.at/	URLElem	A URL to any HTML document. Clicking the URL opens the specified document in an external browser (e.g. Netscape).
	PictureElem	An element displaying a picture of the supported types (GIF, JPEG). Pictures embedded directly in the text are displayed in reduced size and may be shown fullsize in a separate window if desired.
	MediaElem	An element capable of displaying audio and video sequences. Supported formats are MPG, MOV and RMF (These elements where implemented using the Java Media Framework).

Table 1: Predefined Text Elements

5.4. Building Behaviour Diagrams

Behaviour Diagrams show object interaction at run time. A single object is represented by a box which holds the name of the object in its header. The object box wraps one or more method description boxes. These method boxes contain the relevant documentation of involved methods in pseudocode. This documentation is a hypertext with the same capabilities (including text elements) as the previously described hypertext documents. Additionaly a new text element ➡ is introduced, which allows to connect boxes via arrows. A connection between two boxes represents the sending of a message to another object (see Figure 21). This so called *LinkElem* can be inserted at the caret position via menu entry or by pressing the F1 key.

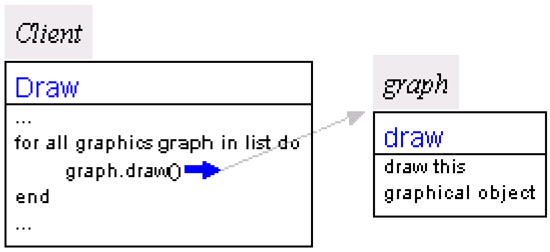


Figure 21: Send the Message 'draw' to the Object ‚graph‘

New object boxes are created through a popup menu (clicking the right mouse button on some empty screen space, see Figure 22).

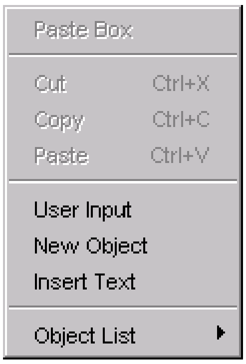


Figure 22: Popup Menu

The first non-inverse entries are used to create different box tyes:

- `User Input:` create a simple text box representing user input
- `New Object:` create an empty object box.
- `Insert Text:` create an empty textbox

`'Object List'` shows a list of all objects existing in the diagram. Selecting one of these objects displays this object directly without navigating through all connection arrows pointing to this object.

If an object is right-clicked, a context sensitive menu pops up, which offers the currently available operations on the clicked object:

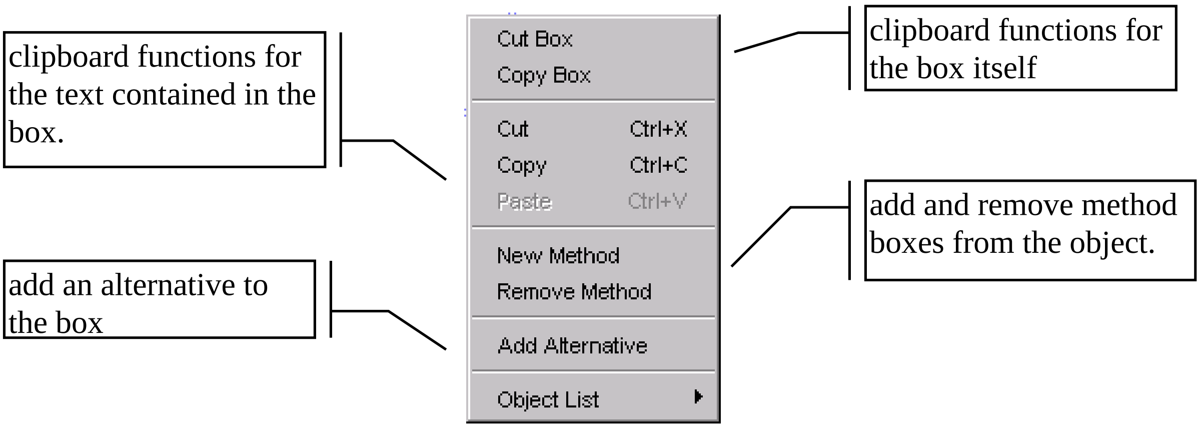


Figure 23: Popup Menu of an Object

5.5. Handling Objects and Links

If a LinkElem is inserted via a menu entry then only the element itself is created and inserted into the text (in this case it holds no connection to another object; this connection is created when activating the element for the first time). If the element is inserted by using the F1 key, an empty target object is created and connected to the new LinkElem. The caret is positioned into the newly created object. This enables the user to draw message flow information very fast without using the mouse.

Links may be redirected by dragging the connecting arrow with the mouse to another target. Links may also refer to methods of the source object itself (these links are called *self references*).

Just like links, also an object may be positioned with the mouse by left clicking at its border and dragging it to the new position. This position is held by the object until is becomes invisible. Each time an object becomes visible (when loading a diagram or following a link), S.I.D. recalculates its position. This automatic layout is necessary to avoid overlapping objects and to reduce the amount of screen space needed by the diagram. Pressing the F5 key enforces an update of the layout.

To save additional screen space, methods can be displayed in a collapsed state showing only the method header (When a method is collapsed its border turns red). A single method is collapsed by clicking on the dividing line between method header and method body (see Figure 24). Collapsing closes all links coming out of the method.

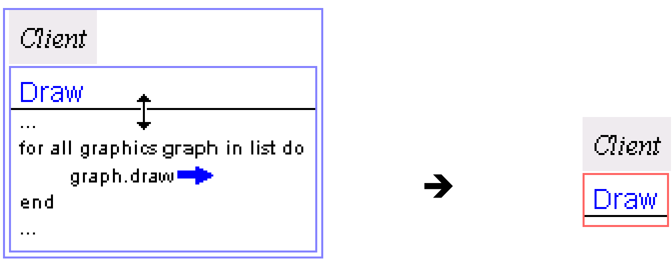


Figure 24: Collapsing a Method

Choosing menu item `'Edit/Collapse All Methods'` collapses all methods with no outgoing links. `'Edit/Expand All Methods'` expands all collapsed methods.

5.6. Handling Alternatives

In order to show dynamic binding, it is necessary to offer more than one special target object when following a link. This problem arises when using Java interface variables or a reference of the baseclass type of certain concrete subclasses. In this case the dynamic type of the target object may vary. In S.I.D. dynamic binding is represented by 'Alternatives'.

Each object box may contain alternative object descriptions representing different subclasses of a given superclass. By choosing between these alternative desciptions, the user plays the role of the dynamic binding mechanism at run time. Since each object is a possible source for other connections, different paths through a behaviour diagram are possible.

In the popup menu of an object a new alternative can be added by the user. This alternative completely covers the current object (see Figures 23 and 25). The alternative contains the same methods as the clicked object although these methods are empty except of the method header. Objects that contain alternatives are marked by an additional border (a styled stack of sheets). Only one alternative of a given set is visible at a time.

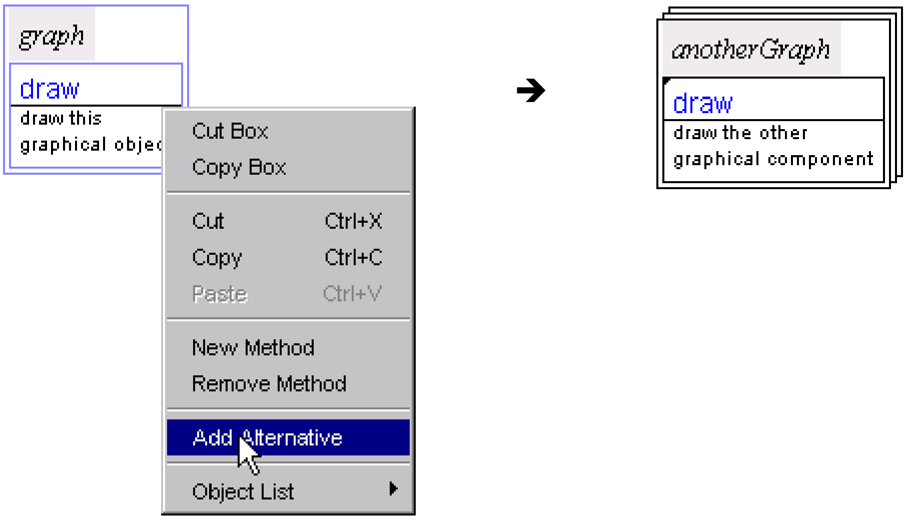


Figure 25: Adding and Editing an Alternative

Methods that are part of the base class or the interface, are called *common methods*, they are marked with a small black spot in the upper left corner of the method box (see Figure 25). If an open link references a common method and the user switches to the next alternative of the referenced object, the link stays open, since it references a part of the interface. This open link represents a dynamically bound call of the referenced method. If a link references a *private method* (a method that is not a common method) and a switch happens, this link is closed, since the referenced method is no longer availabe in the new object.

The popup menu of a set of alternatives contains several additional entries:

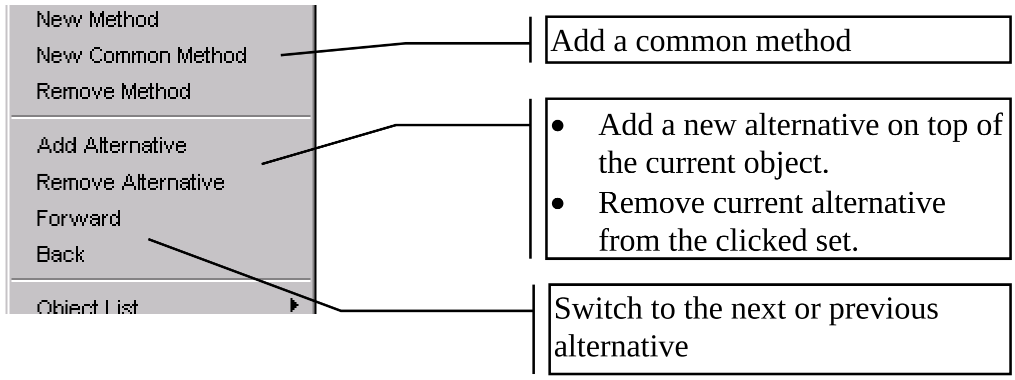


Figure 26: Menu Entries for Alternatives

New common methods may be inserted as well as methods not occurring in the interface. If a common method is added, it is actually inserted into each object description in this set. Methods may be deleted. Deleting a common method in one alternative removes the common mark in all other alternatives of the set.

5.7. Different Forms of Hyperlinks

In S.I.D. it is possible to link Hypertexts and Behaviour Diagrams. Hyperlinks consist of two parts: *MarkElems* are the target of *HyperlinkElems* (see Table 1).

Figure 27: HyperLinks-Menu

Insertion of a hyperlink is implemented as a clipboard mechanism. A newly inserted MarkElem (`Insert Hypertext Target`) is inserted into an internal clipboard, too. The MarkElem may also be copied by hand (`Copy Target`). With this clipboard as source the user may insert as many anchors (HyperLinkElems) to this target as desired (`Paste As Link`), also see Figure 27.

The internal linking mechanism uses text elements and is therefore not capable to connect to external documents like HTML or ASCII files. HTML is a de facto standard for hypertext documents and therefore supported by URLElems. These elements hold a URL and an optional description string. Clicking them connects to the URL and shows the HTML document in an external browser. The path to the browser is inserted in the Options-Dialog (menu `Project/Options` in the main window).

Another special form of hyperlink are links directly into source code stored in the ASCII-format (also see Chapter 4.4.). Such hyperlinks are represented by *SourceLinkElems* (see table 1). Documentation files may be copied to the S.I.D. documentation directory in order to keep them local and consistent. This is no solution for source code, since sources are usually maintained on a central server to be accessed by the different developers. Thus source code links have to be capable of referencing documents on remote servers. S.I.D. uses the FTP-protocol for remote access to source files. This requires a FTP server to be installed on the host where the sources are maintained. Usage of the FTP protocol has the advantages of access control on the server and the high availability and reliability of FTP server software (even on the Shareware market).

When creating a link to a source file, a dialog is opened, which builds up the connection to the desired file (see Figure 28). This dialog transparently handles local and remote files by providing a simple FTP client. The host map on the left side is kept in a file named 'hosts.dat' which is global to the entire application.

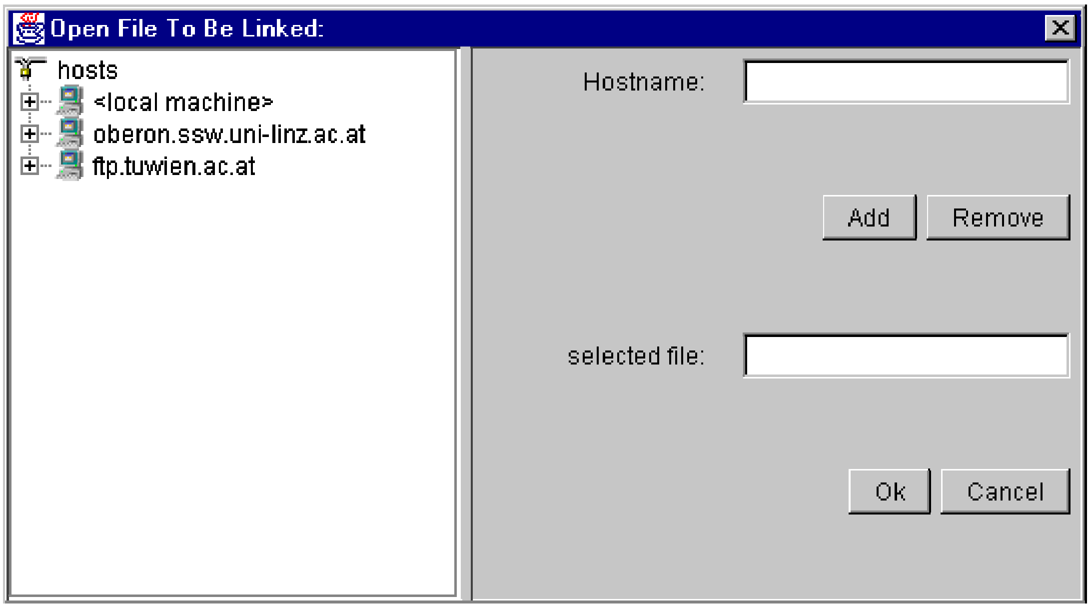


Figure 28: Dialog for Remote Connection

In the subsequently opened textframe, the position in the source file is selected by setting the caret on the desired position (this is supported by a keyword search and a method positioning mechanism, see Figure 29). In the menu only those methods are displayed that exist in the source file (no inherited methods).

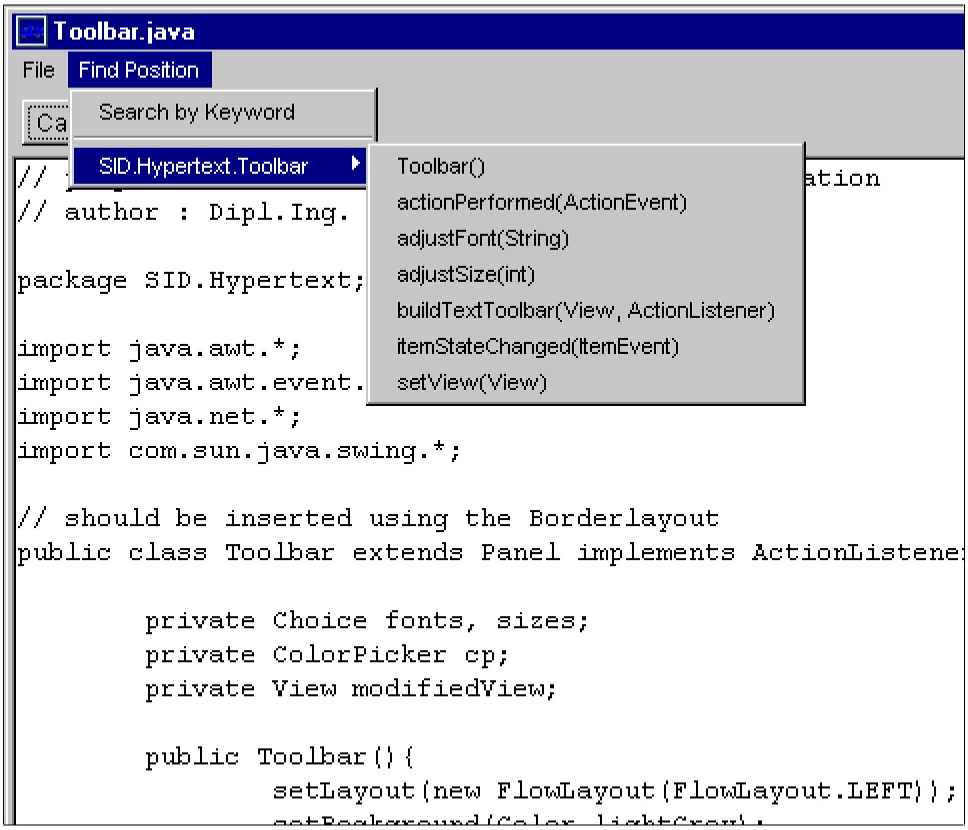


Figure 29: Positioning in the Source File

6. Bibliography

[Cyb92] Cybulski, Jacob: *A Hypertext Based Software Engineering Environment*, IEEE Software, March 1992.

[Dev90] Devanbu, Premkumar, et.al.: *LaSSIE: a Knowledge-based Software Information System*, ICSE 1990.

[Fow97] Fowler, Scott: *UML Distilled - Applying the Standard Object Modeling Language*, Addison-Wesley 1997.

[Fre96] French, Knight, Powell: *Hypertext Structures and Software Documentation*, Technical Report CS-96-04, Department of Computer Science, University of Virginia.

[Gam95] Gamma, Helm, Johnson, Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[Gar90] Garg, Pankaj: *A Hypertext System to Manage Software Life-Cycle Documents*, IEEE Software, May 1990.

[Hor86] Horowitz, Ellis, Williamson: *SODOS: A Software Document Support Environment – It's Definition*, IEEE Transactions on Software Engineering, vol. 12, no. 8, August 1986.

[Jac94] Jacobson: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley 1994.

[Joh92] Johnson: *Documenting Frameworks using Patterns*, OOPSLA 1992.

[KM95] Koskimies, Mössenböck: *Scenario-Based Browsing of Object-Oriented Systems with Scene*, ICSE 1996.

[Pre95] Pree: *Active Guidance of Framework Development*, Software - Concepts and Tools 1995.

[Pre96] Pree: *Framework Patterns*, SIGS Management Briefings 1996.

[Meu97] Meusel, Czarnecki, Köpf: *A Model for Structuring User Documentation of Object Oriented Frameworks Using Patterns and Hypertext*, ECOOP 1997.

[MK95] Mössenböck, Koskimies: *Active Text for Structuring and Understanding Source Code*, SOFTWARE - Practice and Experience, July 1996.

[Rum91] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen: *Object-Oriented Modelling and Design*, Prentice-Hall 1991.

[Wei88] Weinand , Gamma , Marty: *ET++ - An object-oriented Application Framework in C++*, OOPSLA 1988.