CHAPTER 2

The Mjølner BETA system

Peter Andersen, Lars Bak, Søren Brandt, Jørgen Lindskov Knudsen, Ole Lehrmann Madsen, Kim Jensen Møller, Claus Nørgaard, Elmer Sandvad

The Mjølner BETA System is an *integrated* and *interactive* general-purpose software development environment that supports industrial strength programming, using object-oriented programming in the BETA programming language. The aim of the environment is to support the entire software development process, including object-oriented design and implementation, user interface construction, persistence, database construction and maintenance, graphics programming, distributed systems, and documentation.

The integration of the various tools in the Mjølner BETA System is established by insisting that all tools in the system utilize one single representation of the program in the form of abstract syntax trees (ASTs). All manipulations of the ASTs by the various tools are done by utilizing the metaprogramming system, which defines an interface to the AST and ways to manipulate the AST (as illustrated in figure 2.1).

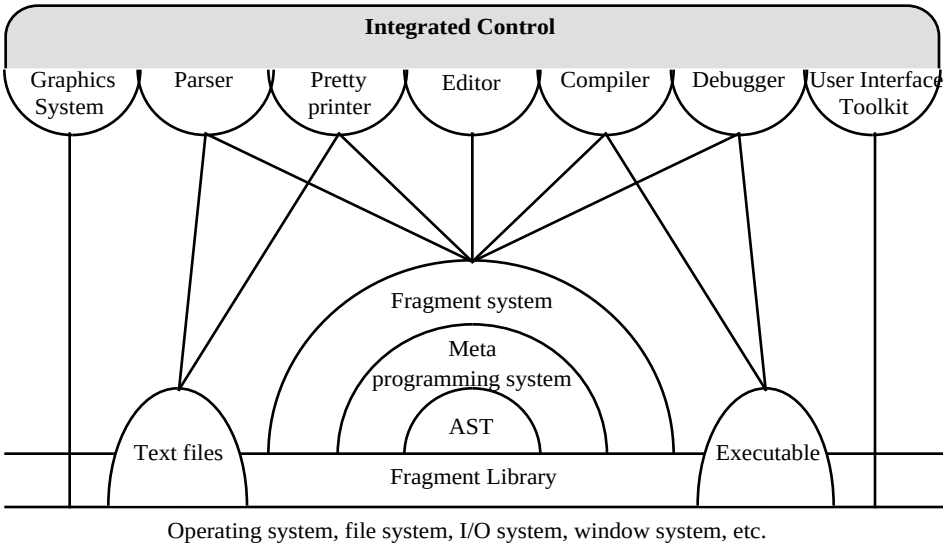

Figure 2.1 *The Mjølner BETA System architecture*

The Mjølner BETA System includes an implementation of the BETA language, a series of libraries and application frameworks, a set of development tools, and a metaprogramming system. All components in the Mjølner BETA System are constructed in the BETA programming language (except the run-time system and a few other routines written in C and assembly language).

Major parts of the system (e.g. editor, parser, pretty-printer, metaprogramming system, fragment system) are grammar-based in the sense that tool generators exist that, given a specific grammar for a language, will define a specific tool that is able to manipulate programs written in that specific language. Such language specific tools have been generated for the BETA language, and form the basis for the Mjølner BETA System. Furthermore, the generators have been used to create tools for many other languages.

In the following, we will give a brief introduction to these different aspects of the Mjølner BETA System, focusing on the usages of the different components. Most components will be described in greater detail in later chapters of this book.

2.1 The BETA programming language

The BETA language constructs are directly designed to enable the construction of effective solutions to software construction using “state-of-the-art” object-oriented language constructs. BETA is a modern object-oriented language from the Scandinavian school of object-orientation where the first object-oriented language, Simula (see chapter 5), was developed. BETA is a strongly typed language like Eiffel [Mey89a] and C++ [Str86a], with an optimum balance between compile-time type checking and run-time type checking.

The language consists of a small number of concepts and the power of the language is the orthogonality of these concepts. This implies that the language is easy to learn, since there are few basic constructs to learn, and thus only a small set of rules for combining these constructs. Table 2.1 illustrates some of the mechanisms of BETA. The basic constructs of the BETA language are patterns and objects. Patterns are used to describe classes, procedures, functions, coroutines, processes, exceptions, etc., and objects are the instances of these patterns. That is, objects are used to describe procedure and function invocations, coroutine executions, concurrent or alternating processes, exception occurrences, etc. In addition to the pattern mechanism, BETA has mechanisms for specifying subpatterns, virtual patterns

Table 2.1 *The BETA language matrix*

	class	*procedure*	*function*	*coroutine*	*process*	*exception*
pattern	√	√	√	√	√	√
subpattern	√	√	√	√	√	√
nested patterns	√	√	√	√	√	√
virtual pattern	√	√	√	√	√	√
pattern variable	√	√	√	√	√	√

and pattern variables. Furthermore, BETA defines means for describing whole/part objects, reference attributes, and general block structure. The BETA language is described in more detail in chapter 6.

2.1.1 The fragment system

The fragment system is a unique system for handling issues related to separate compilation, modularization, information hiding, variant handling and separating of interface and implementation. In contrast to most other modularization and separate compilation systems, the granularity of modularization and separate compilation is controlled by the programmer. The basis of the fragment system is that the programmer defines which parts of the program should be separated into distinct modules (called fragments). This implies that the fragment system allows the modularization that fits the problem at hand, resulting in more elegant and intuitive modularization. The fragments are specified by the programmer, including specification of the interdependencies between the fragments, such that the compiler is able to avoid compilation of already compiled fragments and only recompile these fragments if they have been changed since the last compilation. This implies that the software development is relieved from maintaining huge make-files, specifying the interdependencies between files.

The flexibility of the fragment system is also important in relation to information hiding and separation of interface and implementation. Since the programmer is in control of the granularity of the fragments, he is thereby also in control of the information hiding and separation of interface and implementation. This implies that the programmer may choose to define the information hiding as demanded by the problem, since he may choose to take a mixture of patterns and objects, and view them as one module (or subsystem) with joint information hiding (e.g. only one or two of the components visible to the outside), or he can choose to make each component (pattern or object) into a module, controlling the visibility concerning each component individually.

The fragment system is also a valuable tool for the specification of program variants, especially for maintaining variants for different machine types (on the same file system). The compiler actually uses this part of the fragment system to automatically select the proper variants when compiling on the different machine types. The same facility can be used for cross-compiling onto another machine. The fragment system is described in more detail in chapter 9.

2.1.2 The BETA compiler

The BETA compiler is an effective implementation of the BETA language. The main components of the compiler are the *semantic analyzer* and the *code generator*. The semantic analyzer checks the correctness of the context sensitive syntax (static semantics) of an AST, and performs storage allocation. The code generator translates an AST into executable code (native machine code). The code generator is divided into two components: the *synthesizer* and the *coder*. The synthesizer defines a machine independent model of the code generation, and the coder takes

care of the machine dependent parts of the code generation. The synthesizer is the largest part of the code generator. This implies that porting the compiler to another machine can be done with a reasonable effort. A symbol table is constructed during semantic analysis. The symbol table is defined by means of the semantic level of the metaprogramming system, i.e. the AST decorated with semantic attributes. In this way the symbol table information is an integrated part of the AST and thereby available for other tools accessing the AST (e.g. the editor). In order to manipulate the ASTs, the compiler makes extensive use of the metaprogramming system. Furthermore, in order to generate ASTs from textual program representations, the compiler makes use of the parser. Finally, the compiler makes use of the pretty-printer to generate a textual representation of parts of the AST (e.g. in order to indicate program errors). The compiler uses the fragment system to enable programs to be divided into smaller fragments for separate compilation. The compiler makes an automatic dependency analysis on the fragment structure. When a fragment has been changed, the system keeps track of the dependent fragments that must be recompiled.

The Mjølner BETA compiler generates native code for the target machine, and the BETA runtime system takes care of the allocation of objects and the management of the execution of the actions associated with the individual objects. The runtime system realizes automatic garbage collection of objects when they are no longer reachable by any other object. The compiler also offers powerful means for interfacing to external code and data specified in assembly, C and Pascal. This implies that a BETA program may invoke external functions and access externally allocated data structures. On the other hand, BETA objects may be invoked or manipulated from the external code.

2.2 Libraries and application frameworks

The Mjølner BETA System includes a series of libraries and application frameworks for the construction of high-quality programs. These libraries and application frameworks have many predefined functionalities, ready for reuse by the application programmer. These components are aimed at automating many very common programming problems such that the application programmer can concentrate on the problem domain. The libraries include:

Basic libraries including facilities for text manipulation, exception handling, stream-based communication (I/O), file system access, and external language interfaces. These libraries are the very basics of any program. They offer the basic data structures and interfaces to the operating system.

Container libraries is a set of libraries, implementing the most common data structures, ready for use in application programs. The available data structures are illustrated in figure 2.2.

Persistent store implements a fully orthogonal persistence system for BETA objects. The BETA persistence is very simple to use, yet provides type-safe and reliable secondary storage for BETA objects without the application programmer

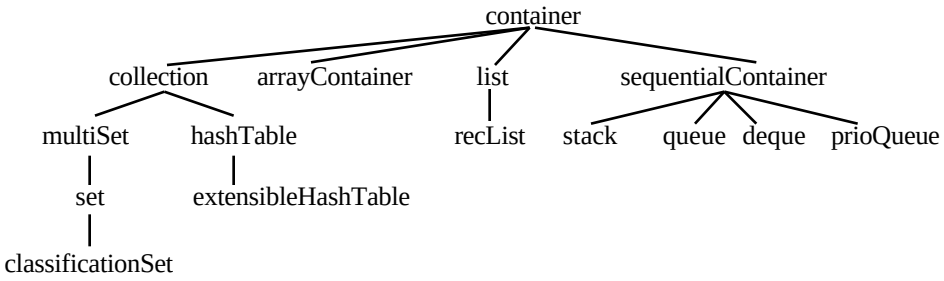

Figure 2.2 *The Containers library hierarchy*

writing any tedious code to transform the BETA objects into some storage form. The only thing the programmer needs to do is to open the persistent store, and mark the persistent root of the objects to be stored in that persistent store. The persistence store will then ensure that the objects are stored on secondary storage in a form that makes it possible for the persistent store to establish the objects again in another program execution (or in another program). When saving (and restoring) BETA objects, the entire transitive closure of the object is stored in (or restored from) the persistent store (i.e. the object is stored along with all objects that are referenced from the object – through references). The persistence library offers an elegant solution to the storage of application data between program executions, or as a communication means between different programs, manipulating some common information structures. The persistent store is described in more detail in chapter 12.

Object browser library implements an elegant ability to realize browsing in objects of the application. If an application program needs to supply browsing facilities for some information structures in the program (such as organizational data in a business application), the object browser library gives one solution. Given an object reference, the object browser library is able to display that object and all objects referenced by that object, using windows to display the individual objects and their state, and the unfolding of object references is totally controlled by the user of the application. Figure 2.3 contains an example of usage of the object browser to display the objects located in a persistent store (in this case, the persistent store named GreatBelt). In the left window, all objects in GreatBelt are displayed (e.g. the object GreatBeltLink) along with information on the type of the objects (e.g. Department). In the right window, the details of the GreatBeltLink object are displayed, such as information on the attribute DistinguishedName, along with the associated value Great Belt Link Ltd.

Concurrency library offers a whole range of pattern abstractions for concurrency in far more structured ways than the build-in language mechanisms (fork and semaphores).

The concurrency library includes patterns implementing monitors with both exclusive access control, and single writer/multiple readers access control. The monitors pattern also includes facilities for medium-term scheduling in the form of a wait-imperative.

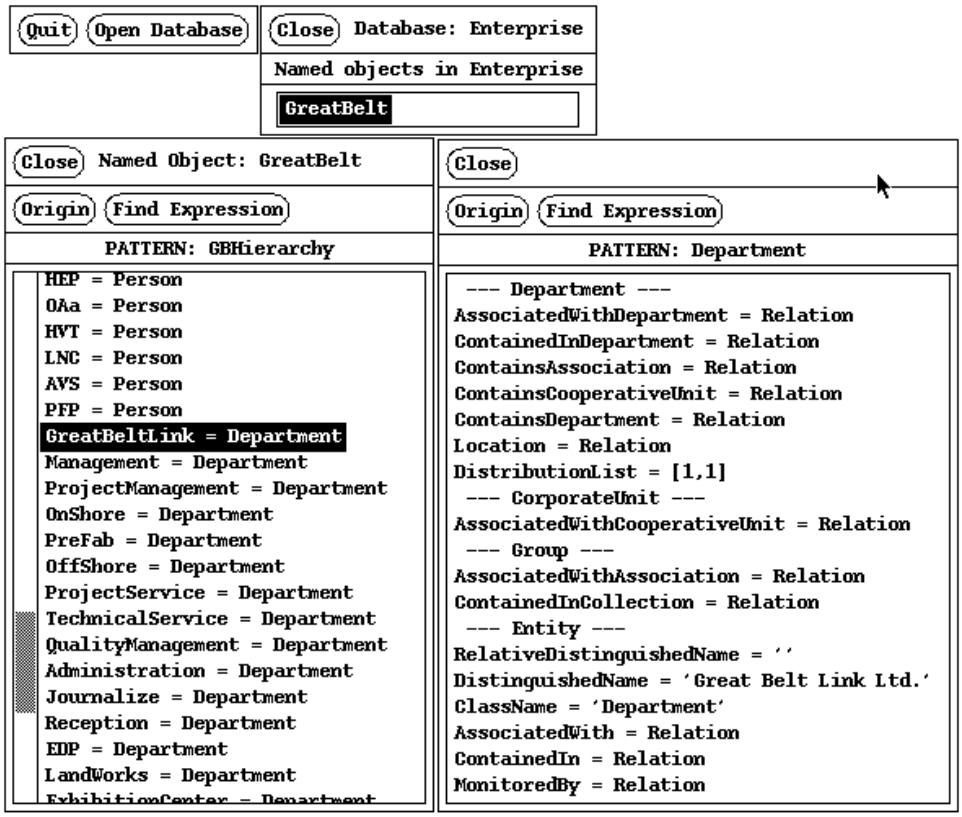

Figure 2.3 *Object Browser windows*

Besides monitor-based communication, the concurrency library includes facilities for rendezvous-like remote procedure communication. This is realized in the form of a port concept, which controls the concurrent access to a set of operations, allowing only the proper objects to invoke these operations remotely (with mutual exclusion). Ports are defined for three often used concurrency control patterns: communication with any other object, communication with objects that are instances of a specific pattern (or a subpattern thereof), and finally communication with a specific object. In all three cases, the ports offer two possible operation semantics: mutual exclusion, or single writer/multiple readers semantics.

The concurrency library is an excellent system for object-oriented concurrent programming, and more importantly, the concurrency control facilities are flexible, allowing the application programmer to define an alternative concurrency control mechanism that suits the needs of the application.

Besides these libraries, the Mjølner BETA System includes application frameworks for helping the construction of advanced software:

XtEnv is an application framework for the construction of user interfaces using the X Window System. The XtEnv framework is based on X Toolkit, and is available in two variants: AwEnv and MotifEnv. AwEnv is based on the Athena widget set, and MotifEnv is based on the Motif widget set. Both implement a BETA object-oriented model, based on the underlying widget-based object model. This implies

that the application programmers should be able to use their insights into these widget sets when programming user interfaces using either AwEnv or MotifEnv.

The BETA object-oriented model of these widget sets is far more elegant than the original C-based object models, and the usage of these frameworks makes the construction of advanced user interfaces easier and more intuitive than when using the C-based object models.

MacEnv is the XtEnv equivalent framework for the Macintosh. In this case, the BETA object-oriented model is designed as part of the MacEnv design effort, and therefore not based on any existing object model. MacEnv makes the construction of advanced user interfaces for the Macintosh family of computers fairly straightforward and intuitive. All event handling and other tedious programming tasks, based on the Macintosh Toolbox, are fully taken care of by the MacEnv framework. The XtEnv and MacEnv libraries are described in more detail in chapter 15.

Bifrost graphics library is a device independent, interactive, extensible, and tailorable 2D graphics system based on the stencil and paint imaging model. The graphics system defines an object-oriented model for graphics modeling, interaction with graphics (creation, reshaping, translation, scaling, and rotation), graphics contexts (local, shared and global), and automatic damage repair. The interactive facilities are an intrinsic part of the graphics model, such that every graphics object that may be displayed on the screen is capable of sensing and responding to interaction from the user using the mouse and keyboard. The Bifrost graphics library is designed such that it cooperates with the MacEnv and XtEnv application frameworks. The Bifrost graphics library is described in more detail in chapter 16.

Distributed objects system is an experimental system for distributing BETA objects onto different computers, and enabling transparent remote procedure invocation in the distributed objects. The distributed object model is based on proxy objects (as illustrated in Figure 2.4), and the application programmer only needs to write specialized code for locating the distributed object. When located (and a reference to the proxy object has been established through the distributed object system), the handling of the distributed system is totally transparent.

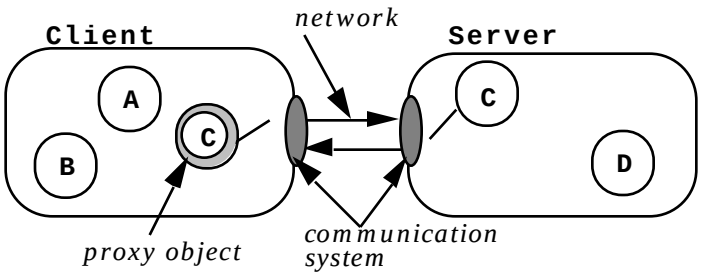

Figure 2.4 *Distributed object system architecture*

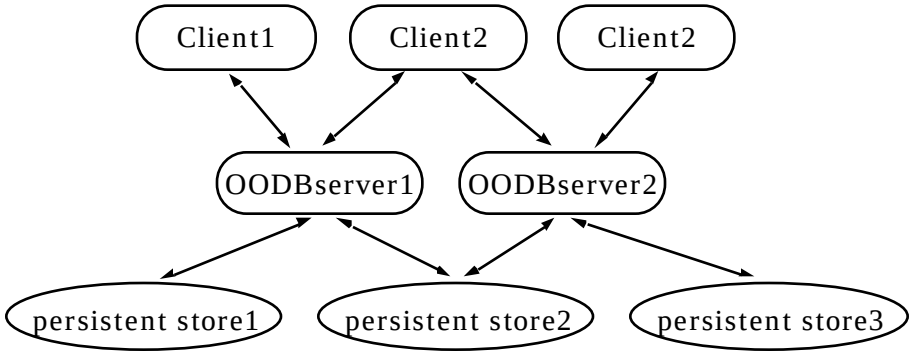

Figure 2.5 *Distributed object-oriented database architecture*

Object-oriented database system for BETA objects has been implemented on top of the persistent store library (and a database engine). The OODB implements nested transactions, concurrency control, localization and query processing in the form of a library of BETA patterns. Furthermore, the OODB defines an extensive notification mechanism, enabling the client to subscribe to modifications in the connected logical databases (such as establishment of other connections to the OODB, creation of new objects, modifications of objects, deletion of objects, etc.).

The database is based on a distributed client/server model in which a client application may have access to several different logical databases (distributed on different database servers), and a database server may control several different logical databases (illustrated in figure 2.5).

2.3 Development tools

The Mjølner BETA System includes a number of development tools, such as source-level debugger, hyper structure editor, and design tool. The Mjølner BETA System allows the utilization of regular text editors as program editors, if the application programmer prefers it. The different development tools are:

Source level debugger for BETA with facilities for setting breakpoints, for browsing in the stacks and heaps of the suspended program, for browsing in the object structure of the suspended program, for browsing in the program structure (patterns and code) of the suspended program, etc.

The source-level debugger is an efficient tool for finding errors in running programs. The facilities for browsing in both the runtime structures as well as the compile-time structures give the application programmer effective means for locating and fixing program errors. Figure 2.6 illustrates the user interface of the debugger. The upper window is the main window, controlling the BETA program being debugged. The OBJECT window displays the current state of an object and the STACK VIEW window displays the current stack. The CODE window displays the code of some object, and in this case the position, where the execution has halted. In any debugging session, several OBJECT and CODE windows may be opened at the same time, giving the ability to inspect the state and code of several objects at the same time.

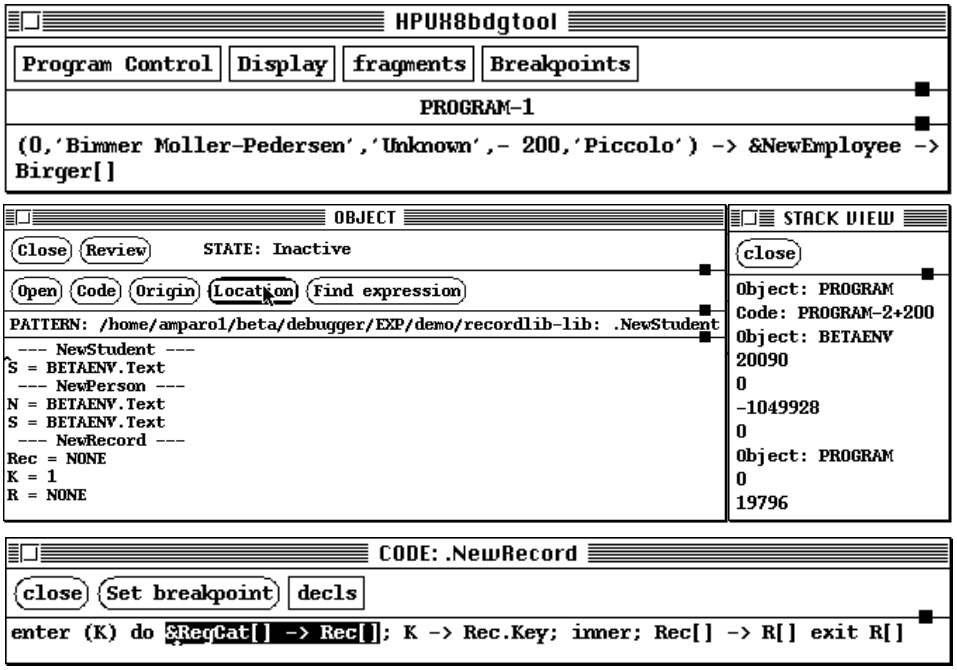

Figure 2.6 *Source-level debugger windows*

Hyper structure editor is the editor for BETA application programming. The hyper structure editor is a structure-oriented editor with free textual editing facilities, such that the application programmer may freely choose to use text editing or structure editing. Immediately after termination of textual editing, the editor will parse the entered (or modified) text incrementally, and report possible parse errors to the programmer to allow him to correct the errors. The editor has a number of other advanced features: *abstract presentation*, *browsing*, and *linking*.

Abstract presentation enables the suppression of details of a given program, such that only placeholders (usually visualized by means of three dots: "...") for this information are shown in the editor window. The suppression of details is totally controlled by the programmer, and gives the programmer total control over which parts of the program are shown in the editor window. The advantage of abstract presentation is that more relevant information can be displayed on the screen at the time, since the irrelevant parts are suppressed by the programmer. Abstract presentation is also applied to comments, since the editor by default merely places a marker where a comment appears in the program. Again this is done in order to keep as much information as possible on the screen at any time, and to encourage programmers to make proper and meaningful comments, which do not hinder the readability of the program during program development. Should the programmer wish to see the comment, he needs only click on the comment marker, and the editor opens a window with the comment, ready for editing. The programmer may also choose to have the comment displayed directly in the editor window as part of the program text.

Browsing enables the application programmer to use the semantic information in the program, e.g. to find the definition of an identifier or find the fragments that a program is dependent on. The browsing facilities include means for opening an editor on the located definitions/fragments, if they are not either part of the current program, or already open. The advantages of these browsing facilities are obviously that the programmer may find information quickly, and spot errors in his understanding of the program (e.g. the identifier is not defined as expected by the programmer).

Linking enables the programmer to make hypertext links within programs, between programs, and in general within and between any document that the editor is capable of handling. Links are bidirectional such that they can be followed in either direction. This general facility can be used by the programmer to link related program pieces for future reference, or for linking a program piece with the documentation of that program piece, thus enhancing the information contents of both the program and the documentation, and to enable quick reference in the future development of this program.

The editor supports *many document types*, since it can handle any document that can be described by a context-free grammar. This includes programming languages and many textual document types, such as reports, documentation, etc., since they are highly structured documents, consisting of chapters, sections, etc. Note that certain document standards (e.g. SGML) actually enable parsing of such documents.

The user interface of the editor is illustrated in figure 2.7. The HPUX8sif window is the main window, controlling the editor, and the two other windows are two editing windows. The right window displays a BETA program, and illustrates the menu for syntax-directed editing, abstract presentation and comment handling. The other window illustrates the editor applied to a structured document, containing the documentation of part of the BETA system. Here also the linking facilities are illustrated (the (^) markers at Streams and Exceptions).

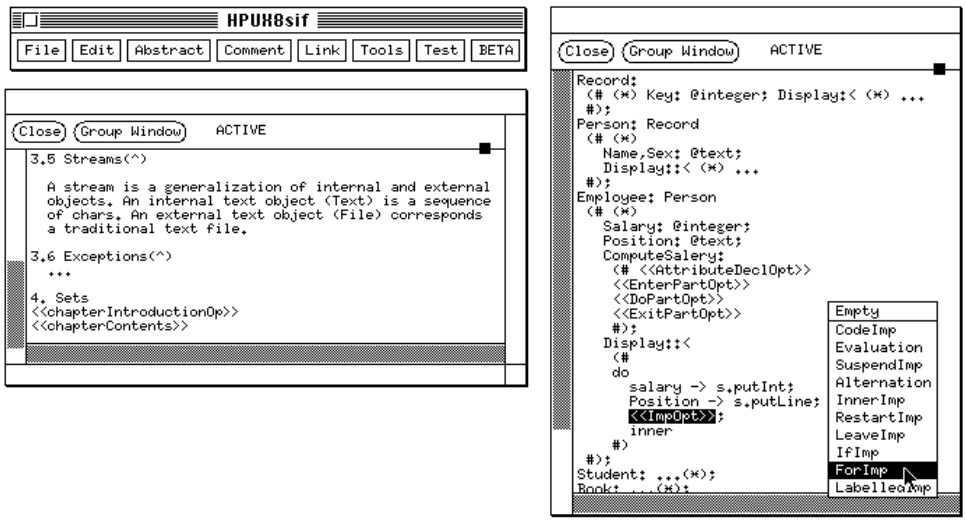

Figure 2.7 *Hyper structure editor windows*

Object-oriented CASE tool is currently being developed for the design and construction of BETA applications in which the usual reverse engineering problems of standard CASE tools do not exist, since all tools work on the same common representation: the abstract syntax trees. The object-oriented CASE tool is a means for creating and manipulating diagrams that graphically view the inheritance structure, the attributes of patterns, and the fragment structure of BETA programs. The hyper structure editor manipulates programs and structured text (e.g. program documentation), and the object-oriented CASE tool manipulates diagrams, exhibiting free vs. structured editing, abstract presentation, browsing, and linking for three different document types: programs, diagrams, and documentation. The linking facility will even work across these different document types. The user interface of the combined CASE tool is illustrated in figure 2.8. The hyper structure editor and CASE tool are described in more detail in chapters 22–24.

2.4 Application-oriented language development tools

The Mjølner BETA System also includes an advanced system for defining languages, such as general purpose programming languages (e.g. BETA, Simula, Modula, Pascal, FelixPascal), specification languages (e.g. OSDL and GDMO), document definition languages (e.g. SGML), database languages (e.g. SQL), and special-purpose application-oriented languages. Examples of such languages are macro languages, scripting languages, and control languages.

A number of tools in the Mjølner BETA System are metaprograms, i.e. programs that manipulate other programs. The Mjølner BETA metaprogramming system is grammar-based in the sense that a metaprogramming tool may be generated from the grammar of any language. All metaprogramming tools in the Mjølner BETA System manipulate programs through a common representation that is abstract

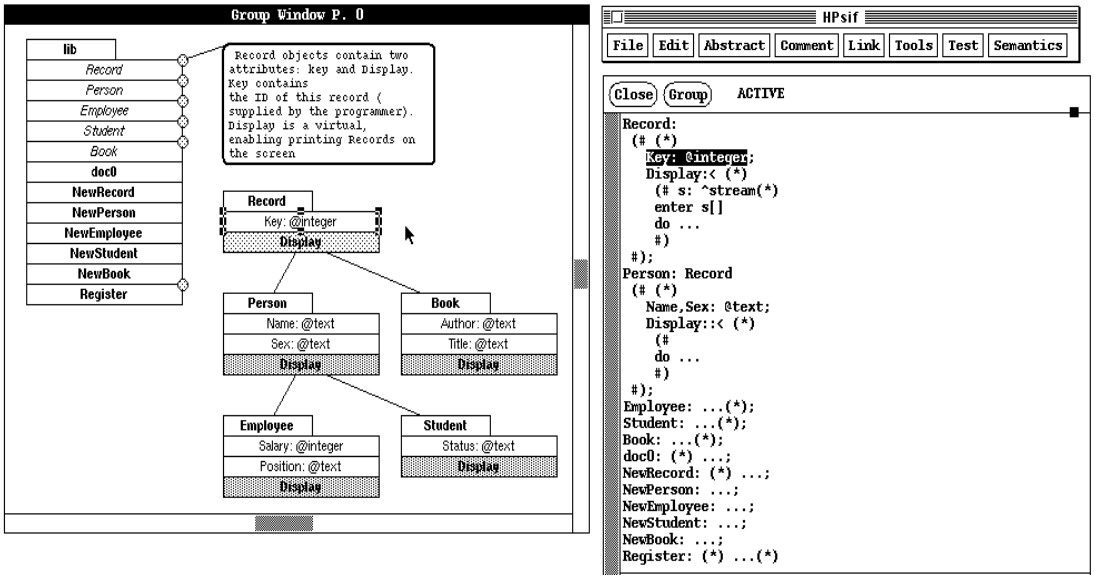

Figure 2.8 *Object-oriented CASE tool interfaces*

syntax trees (ASTs). An object-oriented model of the ASTs has been developed as part of the Mjølner BETA System. An AST is modeled as an instance of a pattern. There is a pattern corresponding to each syntactic category (nonterminal) of the grammar. ASTs derived from a syntactic category are then modeled as instances of the corresponding pattern. The grammar hierarchy is modeled by a corresponding pattern hierarchy. The pattern hierarchy is derived automatically from the context-free grammar.

The grammar-based interface described above results in a set of patterns for each language. A metaprogram using the grammar-based interface will thus be language specific since it uses the set of patterns generated from the grammar of the actual language. A number of tools are language specific in the sense that usually one exists for each language. Examples of tools that benefit from using the grammar-based interface are semantic checkers, program analyzers, interpreters, browsers, graphical presentation tools, and transformation tools.

The fragment system is accessible as part of the Mjølner BETA metaprogramming system, and enables the management and manipulation of coherent ASTs located in different files. The functionality of the fragment system allows splitting of an AST into a number of sub-ASTs (sub-trees) by allowing some inferior nodes in the original AST to be replaced by special nodes. The AST, originally positioned at the position of that node, is called a fragment and may be located in a totally different file.

Two tools exist for converting between textual and abstract syntax tree representations of a program. Both tools are grammar-based and can be applied to any language with a context-free grammar. The *parser* translates a text stream into an AST and the *pretty-printer* translates an AST into a text stream. The parser is based on LALR(1) parsing algorithms. The pretty-printer is an adaptive pretty-printer using a pretty-printing specification to guide the format of the output. For each production in the grammar, pretty-printing directives are given on the layout of sentences, derived from that nonterminal. This specification can be specified by the user. Both tools are used by the compiler and the editor.

In the Mjølner BETA metaprogramming system, an attempt has been made to view traditional tools like the editor, compiler and debugger as metaprograms in general. The advantage of this is that all tools including user programs access programs through a common representation. This leads to the integration of the grammar-based interfaces with the tree level and semantic level described above. The metaprogramming system is described in more detail in chapter 19.