Mjolner Integrated Development Tool − Overview

Mjølner Informatics Report
MIA 91−39
March 2004

Table of Contents

1 Overview

The Mjølner Tool is an integrated development tool with the following components: a source browser and editor, a source level debugger, an interface builder and a CASE tool. The Mjølner Tool is also integrated with the BETA compiler.

1.1 Source Browser (Ymer) and Editor (Sif)

Using abstract presentation and semantic browsing Sif and Ymer provides good overview of libraries and frameworks including your own design and code. Sif is a structure editor. It always operates on complete structures. Sif prevents syntax errors and has powerful operations for modifying even larges pieces of code. The need for scrolling is reduced, making it easy to select large structures not fitting in a window. [Introduction]

1.2 Source Level Debugger (Valhalla)

Valhalla supports breakpoints, single stepping, tracing, object indpection, stack inspection, code inspection, source browsing etc. The accompanying object browser can be used to inspect objects in a persistent store. [Introduction]

1.3 Interface builder (Frigg)

Frigg supports direct construction of graphical user interfaces with automatic incremental code generation and incremental reverse engineering. The developer can alternate between working on the user interface and coding the underlying functionality. [Introduction]

1.4 CASE tool (Freja)

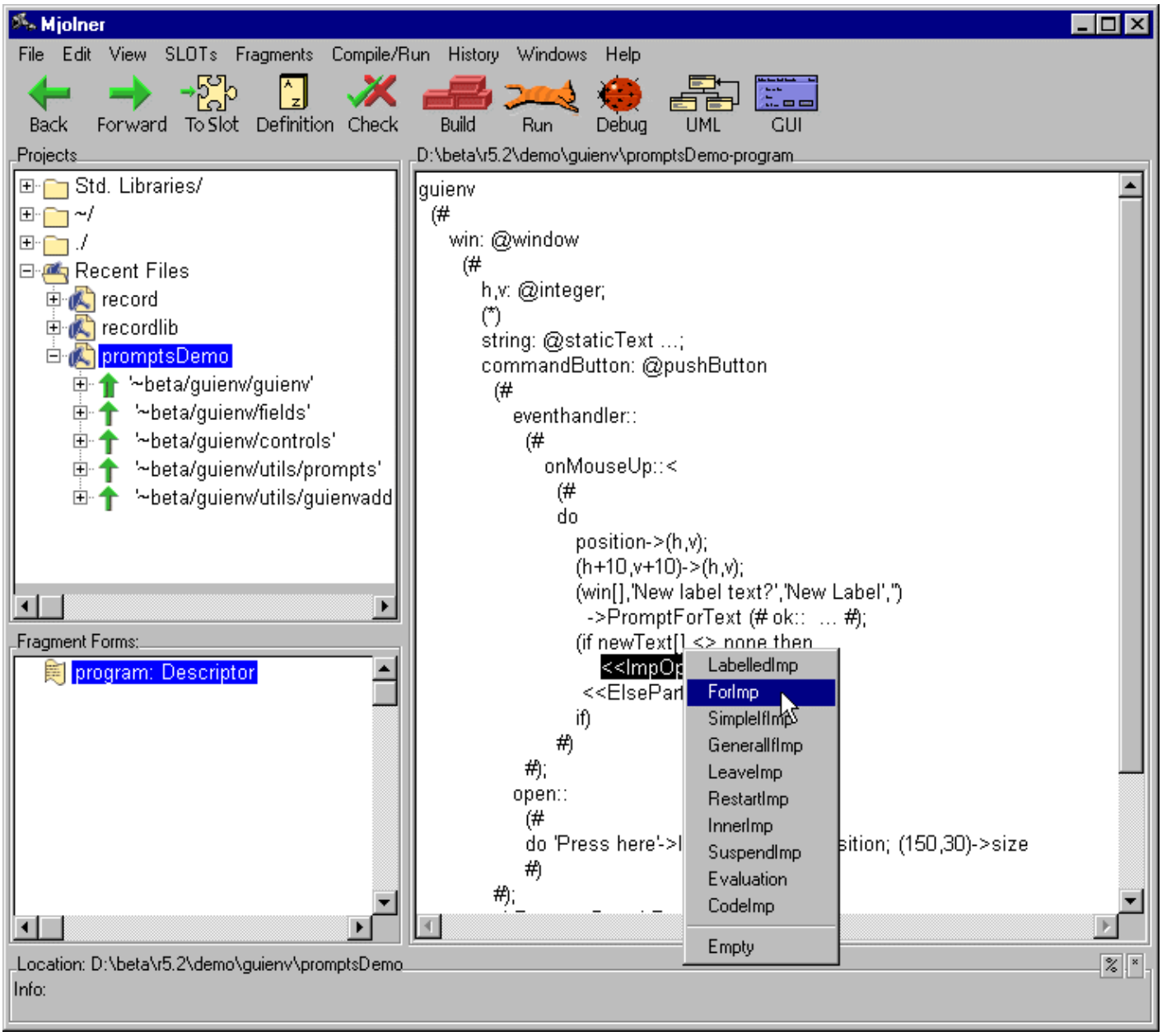
Freja supports the UML−based syntax for BETA. Freja has incremental code generation and incremental reverse engineering. This is achieved without destroying the overview of the code with internal technical comments. Changes in the design are immediately reflected in the code and vice versa. Freja makes it easy to alternate between design and implementation. [Introduction]

2 Source Browser And Editor

Sif [1] is a general grammar−based editor, but it is especially useful for browsing and editing BETA programs. Modularity is in BETA handled by means of the fragment system [MIA 99−42] used for combining fragments into a whole BETA program.

Sif contains a number of basic components: a source browser (Ymer), a fragment group editor and a fragment form editor. Sif is integrated with the BETA compiler that gives a good support for locating and correcting semantic errors.

Figure 1: Source browser and editor



Sif includes the following functionality:

Source Browser

The source browser makes it possible to browse in projects. A project can be a collection of files, a file directory or parts of the dependency graph of a BETA program. Links in the

fragment structure like ORIGIN and INCLUDE can be followed easily.

Fragment Group Editor

The fragment group editor is a high−level editor used to manipulate the dependency graph of BETA programs, e.g. to create and delete whole fragment forms in fragment groups and to create and modify links like ORIGIN and INCLUDE between the fragment groups.

Fragment Form Editor

The fragment form editor is a structure editor that works inside fragment forms. Structure editing is a powerful technique for editing programs without introducing syntax errors. At the fragment form level Sif also provides useful browsing facilities, based on the syntactic and semantic structure of BETA programs.

Structure editing

The basic idea of structure editing is that the program is manipulated in terms of its logical structure rather than the textual elements such as characters, words and lines. The advantage of this approach is that only logically coherent parts can be inserted or deleted and thereby preserving the syntactical rules of the language at any time.
Structure editing is especially useful for application−oriented languages intended for end−users, casual users and beginners that may have difficulties in remembering the concrete syntax. Also a program constructed by structure editing needs not be parsed, thereby saving time in the development phase.

Text editing

Structure editing has its greatest force at the higher levels of editing, i.e. for creating the overall structure of the program or for moving around large chunks of code. At the detailed level the textediting technique is more useful. Therefore the programmer may alternate freely between structure editing and textual editing in Sif. Any program part may be textually edited.

Incremental parsing

Any program part that has been textually edited is immediately parsed. Only the text−edited part of the program is parsed.

Adaptive incremental prettyprinting

The editor includes an adaptive prettyprinting algorithm which prints the program such that it always fits within the size of the window or paper.

Abstract presentation and browsing

The editor is able to present a program at any level of detail. At the top−level of a program the user may get an overview of classes and procedures. It is then possible to browse through the classes and procedures to see more and more details. This mechanism is completely general since the user may decide the level of granularity. Printing a program at different abstraction levels provides a good basis for documentation.

Integration of documentation and comments

The user decides whether or not to display comments. The user also decides whether to display a comment as part of the program or in separate window. A pretty−print of a program which includes just the class and procedure headings (an abstract presentation) and corresponding comments may be produced. This makes it possible to extract an interface specification from the program including the explaining comments.
Programmers are motivated for integrating code and documentation since comments are easy to hide. Large pieces of documentation need not to disturb the overview of the program. Conversely it is easy to extract a high−level presentation of the program including the comments.

Hypertext facilities

The editor includes hypertext facilities. The facility for handling comments is an example of a hypertext link between a program and a text piece. Another type of hypertext link is links from the use of a name in a program to the corresponding name declaration. Such semantic links are very useful especially when working with large programs. At the fragment group level, other examples like ORIGIN and INCLUDE links are links from SLOTs to their corresponding fragment forms and vice versa.

Metaprogramming system

The editor is built upon a metaprogramming system [MIA 91−14], which is available to the user. The user has the possibility to program his or her own metaprogramming tools. It is possible to integrate such tools with the editor or simply to add functionality to the editor. This tailorability is provided by a flexible communication model and the object−oriented implementation language of the editor, which is BETA.

Grammar−basis

The editor is grammar−based, which means that it may support any language that can be described by means of a context free grammar. All the facilities mentioned above (except semantic links which require a semantic checker) are provided for any language that can be described in a context free grammar. Structure editing can not only be used on programs but any kind of documents with a formal or semi−formal structure. In the Mjølner Tool manuals the main focus will be on program editing in BETA. Editors have been generated for programming languages like BETA, SIMULA67, Pascal, Modula−2, for the query language SQL, for the specification languages SDL−92 and GDMO and finally for document types described in subsets of ODA and SGML. In addition the structure editor can be used to create or modify the grammars and prettyprint specifications for each language.

In the Mjølner Tool manuals the focus with respect to editing is on how to use the structure editor for developing programs in one of the already supported programming languages, especially the BETA language.

In [MIA 91−14] it is described how to generate a structure editor for a new language, and briefly how to add functionality to the editor or integrate the editor with another tool.

[Tutorial] [Reference Manual]

---

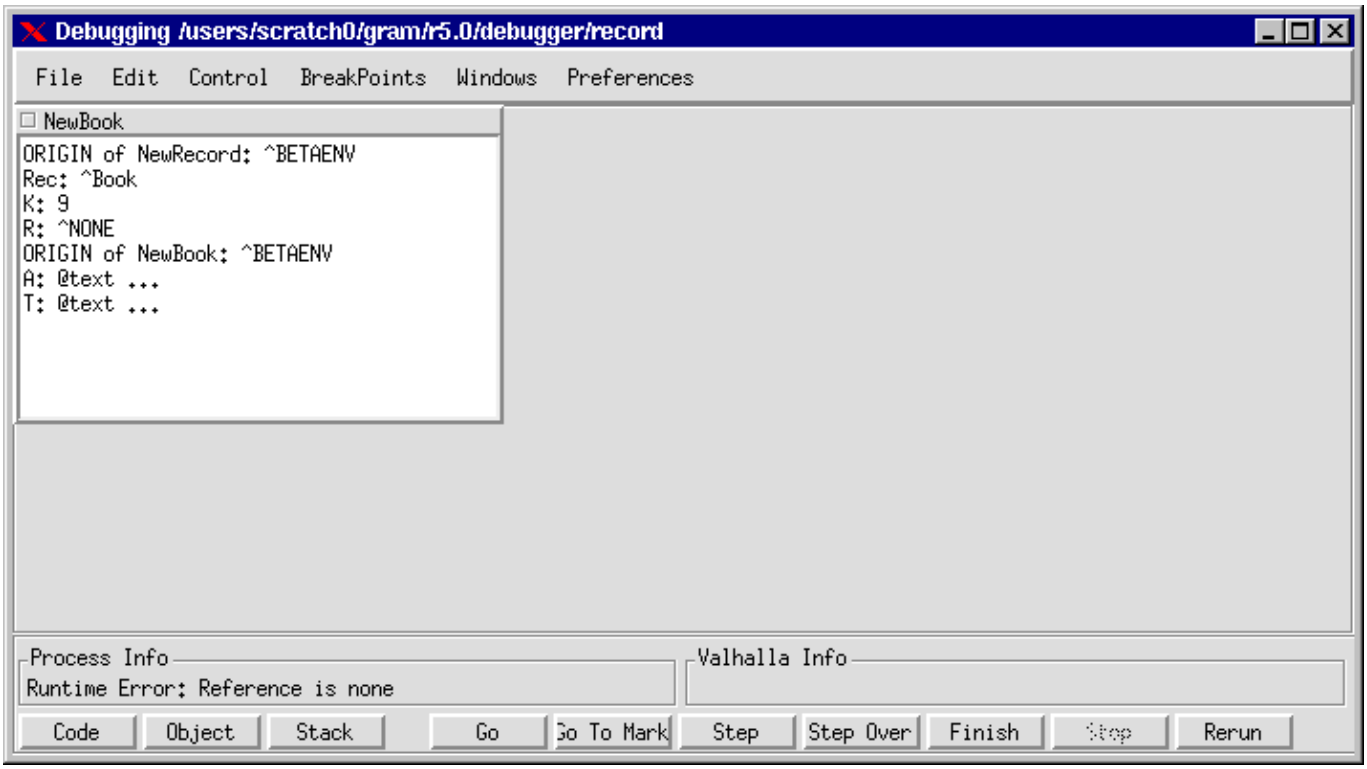
[1] Sif is the wife of the god Thor. Sif is well−known for her golden hair

3 Debugger

Valhalla [2] *is the source level debugger in the Mjølner System. Valhalla offers an object−oriented environment for the debugging of BETA programs. Using Valhalla, you are able to control the execution of any BETA program; inspect the state of runtime objects; and trace the execution of the BETA code implementing the functionality of the application. The aim of Valhalla is to help the programmer to locate errors in BETA programs by tracing their execution at the BETA source level.*

The user interface is divided in two integrated parts: the source browser, and the Valhalla Universe. The source browser enables browsing in the source code of the application being debugged (as well as other source files), and the Valhalla Universe enables the display of runtime objects and execution stacks of the actual execution during the execution of the application.

Figure 2: Debugger universe

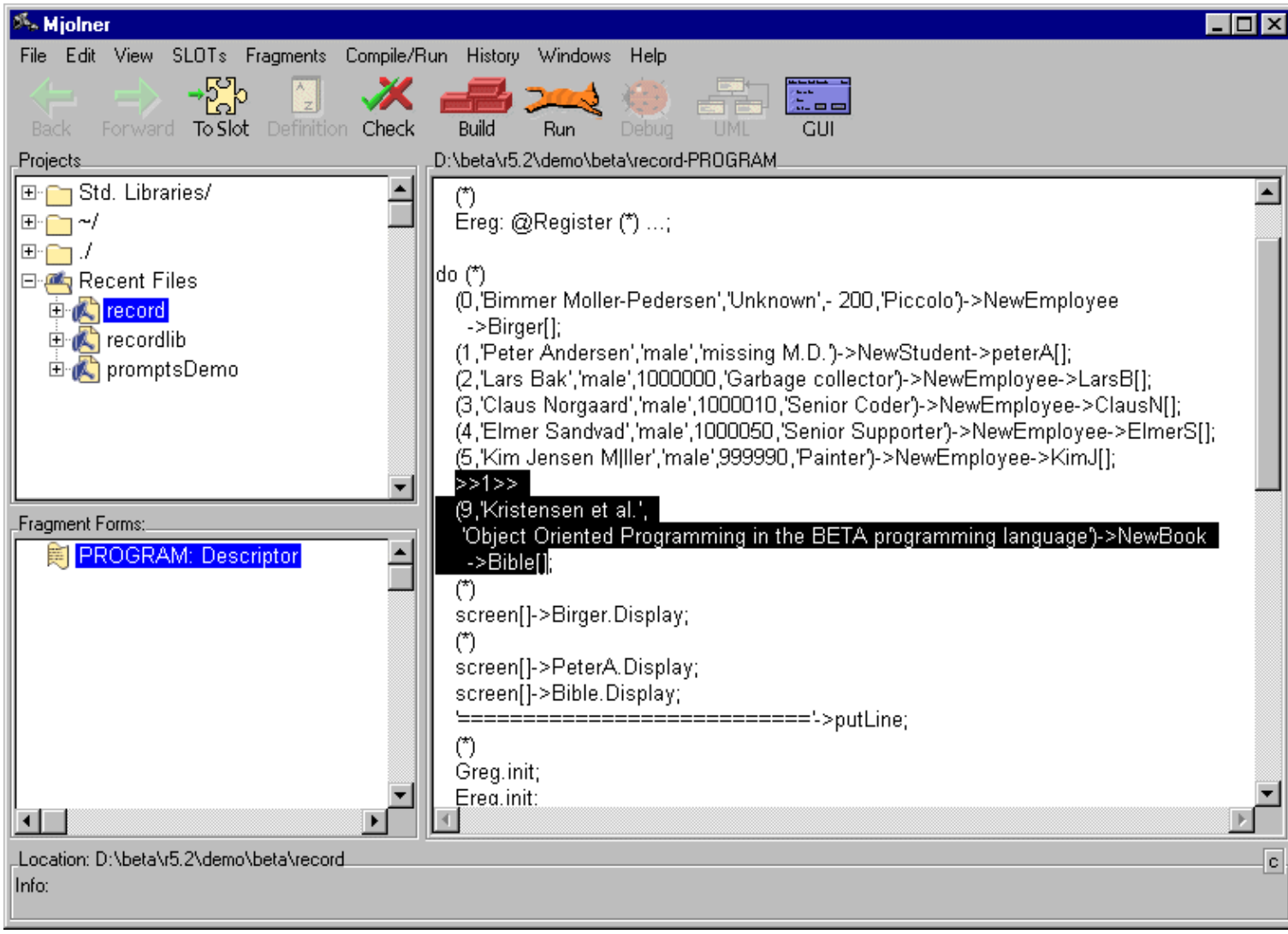


The program execution may be controlled e.g. by setting breakpoints and single stepping at the BETA source level. Runtime errors are caught by Valhalla that will display the offending object and code. From there, program state can be browsed to locate the cause of error. The debugged program executes in a process of its own, being watched by Valhalla, but otherwise unaffected.

Using the Mjølner Debugger you may:

- Control and trace the execution of a BETA program by setting breakpoints and stepping at the level of single BETA source lines, stepping over procedure−calls and even single step at the level of machine code instructions.
- Simultaneously examine the state of any number of objects.
- Examine the execution stack and view code and objects on the stack.
- Examine the program heaps and view objects on the heaps.
- Simultaneously view any number of windows containing source code.

Figure 3: A breakpoint



The breakpoints and current execution point in the code is shown in the sourcebrowser.

Trace points

In order to make it easy to trace the execution of the application, Valhalla offers trace points. A trace point is a point in the source code, with an associated text string. Each time the execution passes a trace point, the text string is printed on the standard output. The text string is specified as part of the specification of the trace point.

Break points

In order to inspect the state of objects during the execution of the application, Valhalla offers break points. A break point is a point in the source code. Each time the execution passes a break point, control is passed back to Valhalla, enabling you to inspect the state of the execution, the state of runtime objects, etc.

Execution Control

Using Valhalla, you can control the execution of any BETA application. You can start and stop the execution, set break points and trace points, single step at the level of machine code instructions, at the level of BETA imperatives, step over procedure calls, etc.

Runtime Inspection

Valhalla offers extensive support for inspection of the runtime structure of the running application. You can examine the state of objects and runtime stacks. Using the easy−to−use graphical interface, you can navigate through the entire object structure, and locate any object in the application, inspecting its state.

[Tutorial] [Reference Manual]

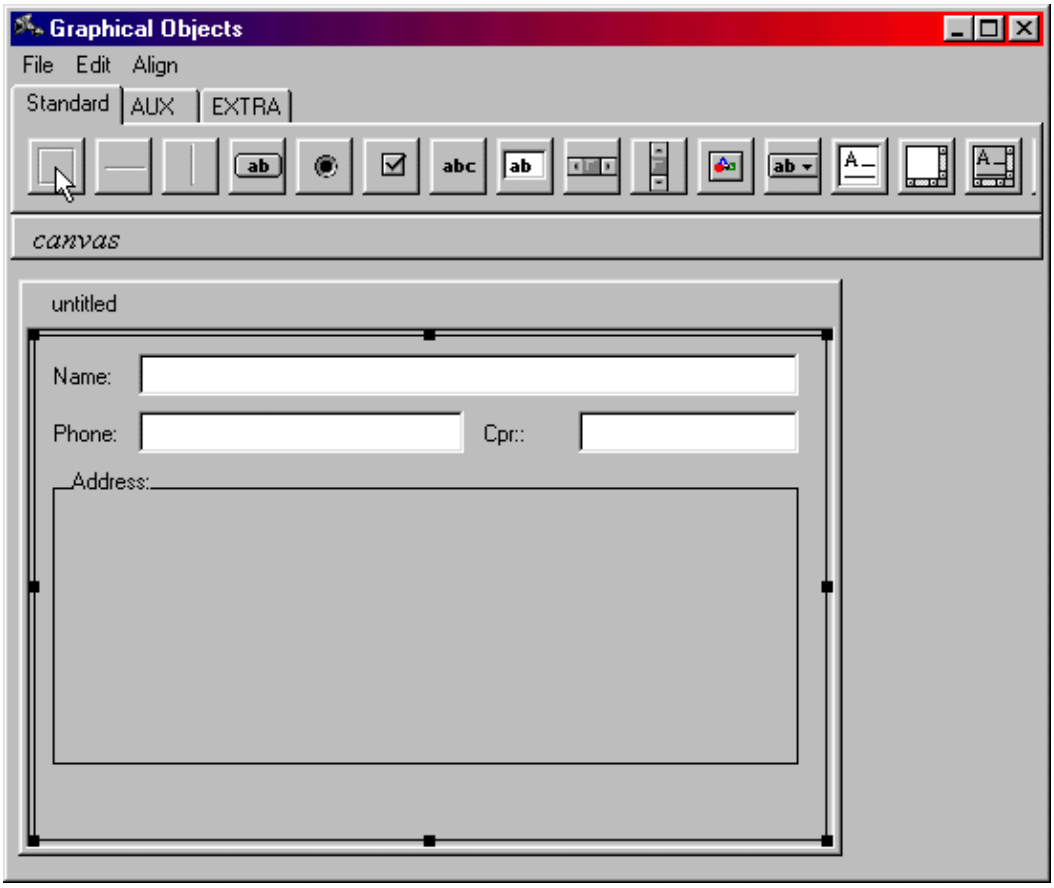
[2] Valhalla is the name of the Hall of the Nordic God Odin. Valhalla is the place whereto all the dead warriors are brought when they have fallen as heroes on the battlefield. Odin is the highest ranking God in Asgaard

4 Interface Builder

Frigg [3] is the interface builder of the Mjølner System, it aims at supporting rapid prototyping and is meant primarily for system designers and developers.

Frigg supplies a graphical editor which interacts with the structure editor to create a prototyping environment. The graphical editor lets its users construct the user interface via direct manipulation, while the necessary code is generated automatically and incrementally behind the scene.

Figure 4: Graphical GUI editor



The generated code can be extended and tailored using the structure editor, and this tailoring can take place at any time during the development of the user interface, because the graphical editor can keep track of the code even though the code has been altered in the structure editor (incremental reverse engineering). The developer can therefore alternate between working on the user interface and coding the underlying functionality. In addition GUI applications developed or modified outside the interface builder can be reverse engineered and modified in the graphical editor.

The integration of Frigg's editors is accomplished using the Mjølner System's uniform representation of programs: abstract syntax trees (ASTs). Manipulations of the ASTs are done through the Meta−Programming System (MPS). Furthermore, all editors working on the same AST are informed when the AST changes, thereby allowing the editors to ensure consistency.

Frigg utilises the Fragment System to: (1) Get access to Lidskjalv, the Mjølner Graphical User Interface environment (GUIenv), (2) Create user interface modules that are properly separated from the model modules of the program, (3) Separate the interface part and the implementation

part of the user interface objects, to allow fine tuning of the users interface without recompiling the entire program.

The code generated by Frigg are specialisation's of the classes in GUIenv.

Frigg does not enforce one particular style of development on its users. In fact, Frigg supports at least the following approaches: (1) starting with design of user interface objects (UIOs) and building up a horizontal prototype having only minimal functionality; (2) starting from a model, and an implemented description of the functionality of a system, but with no UIOs implemented; (3) vertical prototyping, i.e. implementing the functionality behind a subset of a horizontal prototype or fully implementing a small subset of a system intended for incremental extension; (4) simulation of functionality, i.e. adding temporary short cut or dummy computations to a horizontal prototype to support sample data; and (5) full application development. In addition, these different ways of using Frigg can be combined.

[Tutorial] [Reference Manual]

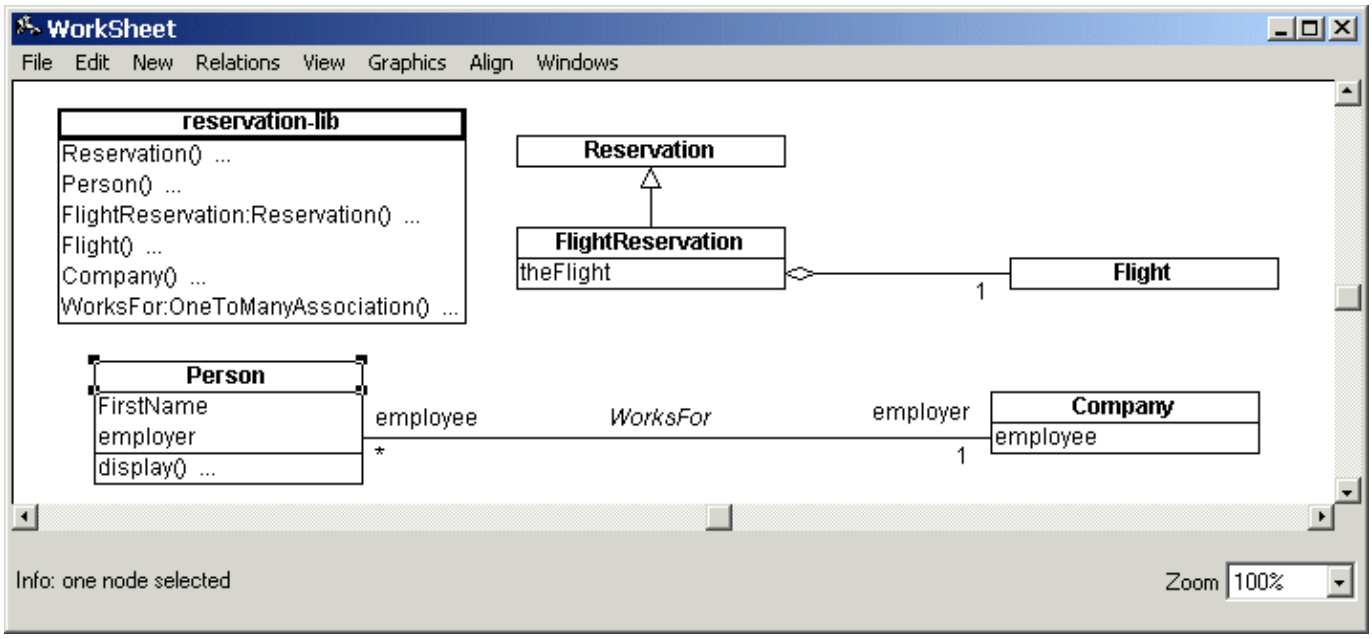
---

[3] Frigg is the wife of Odin

5 CASE Tool

Freja [4] is the CASE tool of the Mjølner System supporting system development with the Unified Modeling Language [UML1.1] and with BETA as the implementation language. Together with Sif (the structure editor), Freja supports a smooth transition from design diagrams to implementation code and vice versa.

Figure 5: Class diagram editor



The CASE tool offers:

Diagram editing
: Design diagrams (class diagrams) can be created and modified.

Automatic code generation
: Code skeletons are generated automatically from the design diagrams.

Reverse engineering
: Design diagrams can be automatically created from the program code.

Simultaneous editing of design descriptions and program code
: The design descriptions and the corresponding program code can be viewed and edited simultaneously, i.e. modifications in the design diagrams are reflected immediately in the program code and vice versa. In other words Freja supports incremental code generation and incremental reverse engineering.

Round trip engineering
: If for example attributes, operations or entire classes are added or deleted outside the context of the CASE tool, the diagram will be automatically updated − with as little disturbance to the existing layout as possible − next time it is opened in the CASE tool.

Freja and Sif are tightly integrated in two ways.

Firstly Freja and Sif share one common representation of the program being developed. The structure editing technique gives a convenient representation, namely an abstract syntax tree (AST). The AST is presented (prettyprinted) textually in Sif and graphically in Freja.

Secondly, they communicate about changes in focus or modifications of the AST. Sif manipulates

the AST through the textual representation and Freja manipulates the AST through the graphical representation. Whenever one of the editors modifies the AST, the other editor is notified and the other representation can be updated accordingly. Since the graphical UML syntax only reflects the overall structure of BETA programs, many modifications in the textual representation may not affect the graphical representation. Conversely, almost every modification of the design diagram, except changing the layout of the diagrams, implies that the textual representation must be updated.

Both representations need not be visible at the same time. In the start of the development process the user might prefer only to see the diagrams, whereas the textual representation becomes more important later on. Since the diagrams and the program are prettyprints of the AST, they can always be reproduced.

[Tutorial] [Reference Manual]

---

[4] Freja is the goddess of love

Index

The entries in the alphabetic index consists of selected words and symbols from the body files of this manual − these are in **bold** font − as well as the identifiers defined in the public interfaces of the libraries − set in regular font.
In the manual, the entries, which can be found in the index are typeset like this. This can help localizing the identifier, when the link from the index if followed − especially in the case where the browser does not scroll the line to the top, e.g. because there is less than a page of text left.
In the small table of letters and symbols below, each entry links directly to the section of the index containing entries starting with the corresponding letter or symbol.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

C

H

I

P

S

V