Teaching Package

Mjølner Informatics Report
MIA 91–17
March 2004

Copyright © 1991–2004 Mjølner Informatics.
All rights reserved.
No part of this document may be copied or distributed without the prior written permission of Mjølner Informatics

Table of Contents

1 Introduction......1

2 Lecture Series 1:The BETA Programming Language......2
2.1 Topics......2
2.2 Lectures......2
2.2.1 Introduction to Object−Oriented Programming......2
2.2.2 Objects, Repetitions, and Patterns......2
2.2.3 Imperatives......3
2.2.4 Subpatterns......3
2.2.5 Virtual Procedure Patterns......3
2.2.6 Block Structure and Part Objects......3
2.2.7 Virtual Class Patterns......3
2.2.8 Pattern Variables......4
2.2.9 Procedural Programming in BETA......4
2.2.10 Co−routine Aspects......4
2.2.11 Concurrency Aspects......4
2.2.12 Non−deterministic Aspects......4
2.2.13 Conceptual Framework for Object−Oriented Programming......4

3 Lecture Series 2: Programming−in−the−large in the Mjølner System......6
3.1 Topics......6
3.2 Lectures......6
3.2.1 The Fragment System (Basics)......6
3.2.2 The Fragment System (Advanced)......6
3.2.3 Exceptional Computation and Overview of Language Constructs......7
3.2.4 Exception Handling......7

4 Lecture Series 3:Overview of the Mjølner System......8
4.1 Topics......8
4.2 Lectures......8
4.2.1 Mjølner System Overview......8
4.2.2 Compiler and Basic Library......8
4.2.3 Containers Library......8
4.2.4 Persistent Objects......9
4.2.5 Distributed Objects......9
4.2.6 Process Communication......9
4.2.7 The Mjølner System Tools......9

5 Lecture Series 4:The Mjølner System Tools......10
5.1 Topics......10
5.2 Lectures......10
5.2.1 Ymer: the Mjølner Source Browser and Editor......10
5.2.2 Sif: the Mjølner Source Browser and Editor......10
5.2.3 Valhalla: the Mjølner Source−level Debugger......10
5.2.4 Frigg: the User Interface Editor for BETA......11
5.2.5 Freja: the CASE Tool for BETA......11
5.2.6 The Object Browser......11
5.2.7 The other tools......11

6 Lecture Series 5:Platform−independent User Interface Construction Using the Mjølner System 12
6.1 Topics......12

Table of Contents

6.2 Lectures 12
6.2.1 Event–based programming 12
6.2.2 Lidskjalv Introduction: Purpose, Overview and Structure of a Lidskjalv Application 12
6.2.3 Lidskjalv Patterns 12
6.2.4 Bifrost Graphics System 13
6.2.5 Frigg: the User Interface Editor for BETA 13

7 Course Assignments 14
7.1 Assignment 14
7.2 Assignment 14
7.2.1 Question A: 14
7.2.2 Question B: 14
7.2.3 Question C: 15
7.3 Assignment 16
7.3.1 Question A Modelling the customer's address 16
7.3.2 Question B: Modelling bank accounts having several owners 16
7.3.3 Question C: 16
7.4 Assignment 16
7.4.1 Question A: 17
7.4.2 Question B: 17
7.4.3 Question C: 17
7.5 Assignment 17
7.6 Assignment 17
7.6.1 Question A: 18
7.6.2 Question B: 18
7.6.3 Question C: 18
7.6.4 Question D: 18
7.7 Assignment 18
7.8 Assignment 19
7.9 Assignment 19
7.10 Assignment 19
7.11 Assignment 20
7.12 Assignment 20
7.13 Assignment 20
7.14 Assignment 20
7.15 Assignment 21
7.16 Assignment 21
7.16.1 Question A: 21
7.16.2 Question B: 22
7.16.3 Question C: 22
7.16.4 Question D: 22
7.16.5 Question E: 22
7.16.6 Tips on using DrawEnv.bet 22
7.17 Assignment 23
7.18 Assignment 23
7.19 Assignment 23
7.19.1 Question A: Hammer windows 23
7.19.2 Question B: Multiple text editor windows 23
7.19.3 Question C: Catching an event in a text editor window 23
7.20 Assignment 24
7.21 Assignment 24

Table of Contents

8 Course Projects....25
8.1 Topics....25
8.2 Project I: Subways in Århus....25
8.2.1 Modelling a subway system in BETA....25
8.2.2 Question A: Placing the subway stations....25
8.2.3 Question B: The shortest path between two subway stations....25
8.3 Project II: Simulation Environment....26
8.3.1 Part 1: Simulation of a group of machines....27
8.3.2 Part 2: Development of general abstractions for simulation....27
8.3.3 Part 3: Development of an interface for simulation....28
8.3.4 Remarks:....28
8.4 Project III: Vehicle Registration Office....28
8.4.1 Remarks:....29

9 Course Materials....30

Index....31
B....31
C....31
D....31
E....31
F....31
I....31
L....31
M....32
N....32
O....32
P....32
S....32
T....32
V....32
Y....32

1 Introduction

This document contains material to be used by lecturers, who are involved in the preparation of courses using the Mjølner System. The intended audience of this material is Bachelor, Master, and Ph.D. level students.

The course material is presented in the form of six Lecture Series involving the Mjølner System in different ways:

- The BETA Programming Language
- Programming−in−the−large in the Mjølner System
- Overview of the Mjølner System
- The Mjølner System Tools
- Platform−independent User Interface Design using the Mjølner System

The Lecture Series are introduced by a general description, a list of topics, and a list of lectures. Each lecture is described by its topic, related readings, related examples, and possibly with additional notes. Any particular course should cover aspects of the first three Lecture Series.

The first Lectures Series focus primarily on the BETA language, and should be accompanied with lectures from the third series of lectures, which contains lectures on the different tools in the Mjølner System such as the compiler, the debugger and the editor. Furthermore, lectures from the second series should be covered since they focus on programming in the large (i.e. issues for modularization of large programs and exception handling. The fourth Lecture Series focus on the tools of the Mjølner System and should be covered, if the students are expected to produce major assignments in the BETA language. The last Lecture Series deals with issues specific for user interface programming using the platform−independent user interface framework

You will find enclosed a number of assignments to be used for the courses. You will also find enclosed a number of project descriptions that are intended to be used for major course assignments or separate student projects. Finally, you will find enclosed a list of books, manuals, etc. to be used during the lectures.

2 Lecture Series 1:
The BETA Programming Language

The purpose of this Lecture Series is to learn the BETA language. The lectures will present all aspects of the language with special emphasis on the language constructs supporting object−oriented programming.

2.1 Topics

- BETA history and BETA language perspective
- Language overview
- Objects
- Patterns
- Attributes
- Enter/Exit
- Object Creation (Static/dynamic)
- Object references (static/dynamic)
- Variables, types, procedures, functions, ...
- Control patterns
- Basic types / basic patterns
- Evaluations
- Scoping
- Subpatterns
- Virtuals (basics/virtual classes(generics))
- Co−routine aspects
- Non−deterministic aspects
- Concurrency aspects

2.2 Lectures

2.2.1 Introduction to Object−Oriented Programming

Readings: [MMN 93, chap. 1, 2], [Knudsen 94, chap. 6]

Notes: It is important to identify object−oriented not just as yet another set of features of programming languages. It is also important to stress that object−oriented programming is not the solution to all software problems. Object−oriented programming is a very important supplement to the other perspectives on programming.

2.2.2 Objects, Repetitions, and Patterns

Readings: [MMN 93, chap. 3, 4]

Notes: Emphasis should be placed on the concept of singular objects, references (static and dynamic), self reference, repetitions and patterns as the general abstraction mechanism. Emphasis must also be placed on the different construction modes for objects (inserted, static and dynamic).

2.2.3 Imperatives

Readings: [MMN 93, chap. 5]

Notes: Special emphasis should be placed on the generality of evaluations, object kinds and construction modes, and computed references and computed remote.

At this time it is appropriate to start on covering the lectures in the third Lecture Series. At least the lecture on compiler and basic libraries. Later in this Lecture Series (when the students begin writing non−trivial programs, it will be appropriate to start on the second Lecture Series on programming−in−the−large. This Lecture Series should be covered before starting the students on one of the major assignments.

2.2.4 Subpatterns

Readings: [MMN 93, chap. 6]

Notes: Emphasis should be placed on patterns and subpatterns as the unique abstraction mechanism for classes, types, procedures, functions, etc. Specifically, emphasis must be places of specialisation of actions, enter/exit parts for subpatterns, qualifications, and the Object pattern. Please emphasise polymorphism in combination with dynamic references.

2.2.5 Virtual Procedure Patterns

Readings: [MMN 93, chap. 7]

Notes: Since specialisation of actions, and the combination with the virtual mechanism is important for the understanding of many libraries, and for the construction of elegant and efficient BETA applications, emphasis must be placed on learning these essential constructs. Remember to illustrate the consequences of combining dynamic references and virtual procedure patterns with enter/exit parts.

2.2.6 Block Structure and Part Objects

Readings: [MMN 93, chap. 8, 10]

Notes: Since block structure is one of the other corner stones of object−oriented programming in the BETA language, emphasis must be placed on learning this essential construct. Especially to students, not previously exposed to block−structured languages. Special emphasis must be placed on multi−nested blocks (patterns within patterns, etc.).

2.2.7 Virtual Class Patterns

Readings: [MMN 93, chap. 9]

Notes: Virtual class patterns are the means for constructing generic abstractions in the BETA language, and leads to many possibilities for elegant modelling.

2.2.8 Pattern Variables

Readings: [MMN 93, chap. 11]

Notes: Pattern variables allows for manipulation of patterns at run−time, giving possibilities for very flexible programming. Please emphasis that pattern variables should be used with care due to their very dynamic nature. Many modelling aspects are best handled by virtual patterns instead.

2.2.9 Procedural Programming in BETA

Readings: [MMN 93, chap. 12]

Notes: Procedural programming is still an important programming style; even after the introduction of object−oriented programming. This chapter illustrates the use of BETA language constructs in procedural programming, and how they augment the facilities for procedural programming as well as object−oriented programming.

2.2.10 Co−routine Aspects

Readings: [MMN 93, chap. 13]

Notes: Naturally, emphasis should be placed on the importance of the modelling capabilities of the action−part as describing the behaviour of components and the attribute part as describing the states of components. Emphasis should also be placed on the construction modes of objects (items vs. components).

2.2.11 Concurrency Aspects

Readings: [MMN 93, chap. 14]

Notes: Illustrates the power of the abstraction mechanisms of the BETA languages, allowing the definition of powerful synchronisation and communication mechanisms to be defined within the language itself, based on a few very basic concurrency facilities (fork and semaphores).

2.2.12 Non−deterministic Aspects

Readings: [MMN 93, chap. 15]

Notes:

2.2.13 Conceptual Framework for Object−Oriented Programming

Readings: [MMN 93, chap. 18]

Notes: Here the underlying conceptual framework for object−orientation is presented. The framework offers a frame of understanding which can be applied to analysis, design and implementation, leading to an understanding of object−oriented analysis, object−oriented design

and object−oriented programming. This material might be split into two lectures, or alternatively also used as some of the justifications for the BETA language constructs.

3 Lecture Series 2: Programming–in–the–large in the Mjølner System

The purpose of this series of lectures is to present the constructs for programming–in–the–large in the Mjølner System. The BETA language does deliberately not include any facilities for modularization, encapsulation, separation of interface and implementation, etc. However, the Mjølner System includes a fragment system which makes modularization, encapsulation, separation of interface and implementation, separate compilation, etc. possible for BETA programs. Furthermore, this Lecture Series will present how exceptions can be dealt with in the BETA language.

3.1 Topics

- Overview of modularization, encapsulation and separation of interface and implementation
- Introduction of grammar–based program modularization
- The fragment system
    - Fragments and slots
    - Origin
    - Include
    - Body
- Modularization using the fragment system
- Encapsulation using the fragment system
- Separation of interface and implementation using the fragment system
- Exceptions
- Overview of language constructs
- Exception handling in BETA

3.2 Lectures

3.2.1 The Fragment System (Basics)

Readings: [MMN 93, chap. 17 (17.1)]

Notes: This lecture presents the general grammar–based approach to program modularization, and its realisation in the BETA fragment system.

It is important to emphasise, that the fragment system is orthogonal to the BETA language. Emphasis that the fragment system controls slots, fragment forms, fragment groups, and the binding of fragments to slots (and the visibility of these bindings).

3.2.2 The Fragment System (Advanced)

Readings: [MMN 93, chap. 17 (17.2–17.6)]

Notes: This lecture discusses the facilities for separating interface and implementation, for making abstract data structures, for creating program variants, etc.

3.2.3 Exceptional Computation and Overview of Language Constructs

Readings: [JLK84,87], [Borgida85]

Notes: Exception handling is an important part of programming large robust systems. This lecture will present the issues involved and a taxonomy for exception handling.

3.2.4 Exception Handling

Readings: [MMN 93, chap. 16]

Notes: Presentation the power of BETA, allowing exception handling to be handled within the language itself.

4 Lecture Series 3: Overview of the Mjølner System

The purpose of this Lecture Series is to introduce the students to the Mjølner System. The lectures will present the BETA compiler, the various basic libraries, the persistent object system, the debugger, and the hyper structure editor. This Lecture Series should be accompanied with one of the following Lecture Series on using the Mjølner System on the specific platform (Macintosh or UNIX).

4.1 Topics

- Overview
- Compiler introduction
- Libraries
    - ♦ Basiclib
    - ♦ Containers
- Persistent Objects
- Distributed Objects
- Process Communication
- The Mjølner System Tools

4.2 Lectures

4.2.1 Mjølner System Overview

Readings: [MMN 93, app.. B], [Knudsen 94, chap. 2]

Notes: This lecture is meant as an introduction to the entire Mjølner System. It presents the general architecture of the system and discusses several of the Mjølner tools.

4.2.2 Compiler and Basic Library

Readings: [MIA 90–2], [MIA 90–8], [Knudsen 94, chap. 26]

Notes: This lecture can be avoided since it only deals with the specifics of how to invoke the compiler, how to invoke BETA applications and how to understand the compiler and run–time error messages. This lecture also describes the layout and facilities of the basic BETA library, that any BETA program utilises. This library offers ordinary data types such as integers, boolean, etc., as well as several useful file access patterns.

4.2.3 Containers Library

Readings: [MIA 92–22]

Notes: The Containers libraries can be used as an example of a library family with extensive usage of the abstraction mechanisms of the BETA language (virtual class patterns and specialised control

structure patterns).

4.2.4 Persistent Objects

Readings: [MIA 91–20], [Knudsen 94, chap. 11, 12]

Notes: The persistent store is designed for storage of BETA objects and is highly useable for "small–scale" external data storage (i.e. not designed for heavy database applications). An object–oriented database system for BETA objects is under development.

4.2.5 Distributed Objects

Readings: [MIA 94–25]

Notes: The distributed object system for BETA is designed and inplemented to enable BETA objects to be located anywhere on a network. Objects located on the network are accessed transparently from the other hosts on the network. Access to distributes objects are obtained through a naming service, implemented by the ensembles, who are the BETA abstractions for network hosts.

4.2.6 Process Communication

Readings: [MIA 94–29]

Notes: The process library implements process communication facilities for BETA programs, enabling them to communicate with other programs using operation system level communication means (such at TCP/IP).

4.2.7 The Mjølner System Tools

Readings: The different man–pages

Notes: In this lecture, you should present the individual tools in the Mjølner System: the source browser and editor, the debugger, the user interface editor, and the CASE tool. If these tools are used extensively in the courses, it would be a good idea to introduce the individual tools more extensively by accompanying this Lecture Series with the Lecture Series entirely devoted to the tools..

5 Lecture Series 4:
The Mjølner System Tools

The purpose of this Lecture Series is to introduce the students to the tools in the Mjølner System. The lectures assumes that the students have been previously presented to the BETA compiler. This Lecture Series will present the source browser and editor, the debugger, the user interface editor, and the CASE tool, along with a few small utility tools. It is usually beneficial if these lectures could be conducted in an auditorium, where the tools at the same time can be demonstrated live using screen projection techniques to make the students follow a live demonstration.

5.1 Topics

- Sif: the Source Code Browser and Editor for BETA
- Valhalla: the Source–lever Debugger for BETA
- Frigg: the User Interface Editor for BETA
- Freja: the CASE Cool for BETA
- Other tools

5.2 Lectures

5.2.1 Ymer: the Mjølner Source Browser and Editor

Readings: [MIA 99–34, chapters on the browser]

Notes: Ymer is the source browser, used in all major Mjølner System tools. Ymer is described in the Sif (see later) tutorial and referce manual, but is available as a componemt also in Valhalla, Frigg and Freja. Using Ymer, you can browse the directory structure and the dependency structure.

5.2.2 Sif: the Mjølner Source Browser and Editor

Readings: [MIA 99–34]

Notes: Use the editor on some program, utilising all editor facilities to get a feel for the different facilities. The advanced features for abstract presentation, browsing and linking should be presented and the facilities for integration of program and documentation demonstrated.

5.2.3 Valhalla: the Mjølner Source–level Debugger

Readings: [MIA 99–34]

Notes: This is a presentation of the BETA debugger. Using the debugger you can control the execution of BETA programs, inspect the runtime stack, and the state of the objects in the debugged program. The object inspection facilities are actually implemented using the object browser (see later).

5.2.4 Frigg: the User Interface Editor for BETA

Readings: [MIA 99–34]

Notes: This is a presentation of the BETA user interface editor. This editor is an extension of the Sif editor, enabling direct manipulation editing of the user interface aspects of the application being edited..

5.2.5 Freja: the CASE Tool for BETA

Readings: [MIA 99–34]

Notes: This is a presentation of the BETA CASE tool. This CASE tools enables you to create design diagrams for your application. Freja is also implemented as an extension of the Sif editor, and in this case with a design diagram editor, in which the design is created, using a graphical design notation, slightly similar to the OMT design notation..

5.2.6 The Object Browser

Readings: The manual page for psbrowser

Notes: The object browser is a library, implementing object inspection facilities. The object browser is used in Valhalla to inspect the objects in the debugged program. The object browser can also be linked into your application, augmenting it with inspection facilities of the objects in the application itself.

A special variant of the object browser exists for inspecting persistent stores. This variant is called the psbrowser. The psbrowser is a very useful toll to introduce to programmers, using the persistent store, since it gives them good facilities for browsing the object structures that are stored in a given persistent store.

5.2.7 The other tools

Readings: The manual pages for these tools

Notes: You might want to introduce the small utility tools: betawc, betatar, and betafs.

6 Lecture Series 5: Platform−independent User Interface Construction Using the Mjølner System

The purpose of this Lecture Series is to learn how design and implement platform−independent user interfaces using the Lidskjalv user interface framework.

6.1 Topics

- Event−based programming
- Lidskjalv
    - Patterns, Fragments
    - Philosophy of Lidskjalv
    - Structure of a application
    - Event handling in guienv
    - Presentation of guienv patterns
- Bifrost Graphics System
- Frigg: the User Interface Editor

6.2 Lectures

6.2.1 Event−based programming

Readings: ...

Notes: If the students have not previously been confronted with event−based programming, they should have a short introduction to this, before starting user interface programming. If they have been used to sequential programming, user interface programming is somewhat 'upside−down', since they are no longer in full control of the sequence of actions being executed in the program, due to the non−deterministic nature of user interfaces.

6.2.2 Lidskjalv Introduction: Purpose, Overview and Structure of a Lidskjalv Application

Readings: [MIA 95−30]

Notes: Lidskjalv is a set of libraries, containing an object−oriented BETA framework for the construction of advanced user interfaces for BETA programs. In this lecture, the architecture of Lidskjalv is presented along with the general approach to handling events etc. from the user interface. Furthermore, the patterns of Lidskjalv are shortly presented.

6.2.3 Lidskjalv Patterns

Readings: [MIA 94−27]

Notes: This lecture will present the various Lidskjalv patterns in details, and present several applications using the guienv interface. The examples are found in e.g. [MIA 95–30].

6.2.4 Bifrost Graphics System

Readings: [MIA 91–13], [MIA 91–19], [Knudsen 94, chap. 16]

Notes: Dependent on the previous graphics knowledge among the students, it might be advisable to split this lecture into two lectures, with the first lecture focusing on the basic graphics model (Stencil & Paint) and the second focusing on the graphics modelling and object–oriented structures in Bifrost.

6.2.5 Frigg: the User Interface Editor for BETA

Readings: [MIA 99–34]

Notes: This is a presentation of the BETA user interface editor. This editor is an extension of the Sif editor, enabling direct manipulation editing of the user interface aspects of the application being edited..

7 Course Assignments

The following is a series of assignments complementing the exercises found in [Madsen93]. Please do not expect these assignments to be complete in any sense. You should see them as proposals, that should be supplemented, depending on the style of the course and the background of the students.

Currently, the order of the assignments is somewhat random, not expressing any levels in difficulty, etc.

7.1 Assignment

Write a BETA program, that converts UNIX file names to Macintosh file names. I.e. converts the string:

```
"/home/saturn/kurt/hello.bet"
```

to:

```
"home:saturn:kurt:hello.bet"
```

Note, that this assignment is based on knowledge about the text patterns in betaenv.

7.2 Assignment

The idea here is to build a search program which, given a line of text, can find occurrences of characters and of other (smaller) strings of text. Here are the relevant commands:

```
T       (* Text: input new "base" text string *)
<text>
C       (* Char: count occurrences of a char *)
<char>
O       (* Occurs: check if some text occurs as a substring (Yes/No) *)
<text>
N       (* Number of occurrences: count occurrences of a substring *)
<text>
F       (* First: first location of a substring *)
<text>
L       (* Last: last location of a substring *)
<text>
Q       (* Quit *)
```

7.2.1 Question A:

Sketch the search algorithms you want to use.

7.2.2 Question B:

Implement a search program (search) in BETA that supports the above commands. You may test your program using the given sample interaction. User typing is printed in underlined. This is only an example – feel free to choose your own style for the interface.

7.2.3 Question C:

How well does your code separate the search functionality from the handling of the user interface? For example, do you think it would be easy to reuse your search patterns in a different environment (say, where the text appears in a window and commands come via menus and dialogues)? Do you have any general reactions to line–oriented interfaces like these?

```
search
Commands:
T. input new "base" text string
C. count occurrences of a char
O. check if some text occurs as a substring
N. count occurrences of a substring
F. first location of a substring
L. last location of a substring
Q. quit

Be sure to end each line of input with <enter>

T
New "base" text string:
abcdefg
Next command:
C
Char to search for:
g
  1 occurrences of "g" in "abcdefg":
Next command:
T
New "base" text string:
abc def ghi def de
Next command:
O
Substring to search for:
f g
Yes - there are some occurrences of "f g" in "abc def ghi def de"
Next command:
O
Substring to search for:
fg
No occurrences of "fg" in "abc def ghi def de"
Next command:
N
Substring to search for:
def
  2 occurrences of "def" in "abc def ghi def de"
Next command:
F
Substring to search for:
de
  5 is the first location of "de" in "abc def ghi def de"
Next command:
L
Substring to search for:
de
 17 is the last location of "de" in "abc def ghi def de"
Next command:
L
Substring to search for:
cd
No location of "cd" in "abc def ghi def de"
Next command:
N
```

```
Substring to search for:
 d
   3 occurrences of " d" in "abc def ghi def de"
Next command:
Q
So long!
```

7.3 Assignment

This assignment is a review of concepts like dynamic references and repetitions. Consider the bank example in [Madsen93].

7.3.1 Question A Modelling the customer's address

Change the address attribute of Customer from a simple text attribute to be a structured object consisting of three text attributes: streetAndNumber, postalCode, city. That is, in the object descriptor of Customer, instead of "Address: @text", you'll have "Address: @(# ... #)". Also, give Address its own Print attribute which can print the three parts of an address. The Print attribute for Customer should (among other things) now invoke the print for Address. You'll probably want to change the Initialize pattern of Customer to set the new Address fields.

The Account pattern shouldn't need to be modified. Again, test your code by creating an account and a customer and printing the balance. Do you agree that Address should be declared this way instead of as a dynamic reference to a separate object?

7.3.2 Question B: Modelling bank accounts having several owners

An account might have several owners (e.g. husband and wife). Change the owner attribute to "owners" and make it be a repetition of dynamic references to Customer. PrintBalance should print the information of all customers before printing the account balance. You'll need a way to add new owners, say, an AddOwner procedure attribute.

Test your code by creating several customers for the same account and printing the balance.

7.3.3 Question C:

Modify the bank system such that it includes a procedure pattern that deletes the account of a specific costumer.

7.4 Assignment

You must design and implement a very simple soda machine which "dispenses" just two kinds of drinks, cola and juice.

Your soda machine should include a current total of money. (You can assume you'll only get 1 dollar coins or larger so it can be an integer.) You must manage two other totals: the current number of colas and the current number of juices. Your soda machine should support transactions where the user puts in money and gets back a cola or a juice plus change. But you'll also need to support the soda machine maintainer who refills the machine and checks the current totals.

Here's a sample user interaction which your program should be able to support. ("R" stands for refill, "C" for buy a cola, "J" for buy a juice, "T" for print totals, "Q" for quit.) What the user types are underlined. Colas cost 8 dollars and juices 7 dollars each.

```
T
Current totals: 0 colas, 0 juices, 0 dollar.

R
How many colas to add?
15
How many juices to add?
20
How many dollars should we start with?
100

T
Current totals: 15 colas, 20 juices, 100 dollars.

C
A cola costs 8 dollars.  How much money are you inserting?
10
Thanks.  Your change is 2 dollars.

J
A juice costs 7 dollars.  How much money are you inserting?
20
Thanks.  Your change is 13 dollars.

T
Current totals: 14 colas, 19 juices, 115 dollars.

Q
Goodbye.
```

7.4.1 Question A:

Identify the objects in the example. What patterns will you need to define? What attributes will they have?

7.4.2 Question B:

Implement the soda machine in BETA and test it.

7.4.3 Question C:

What happens in your program if the number of colas goes down to zero before a refill? What happens if the user puts in less money than the drink costs? Suppose you needed to change the costs of colas and juices? Are those numbers easily adjustable or are they "hard–wired"?

7.5 Assignment

Make a BETA program, which reads a number of numbers (terminated by 0), and outputs the largest and smallest number.

Then modify the program such that it outputs all numbers sorted (use whatever sorting method you want).

7.6 Assignment

Consider the following BETA program:

```
(# Ptn: (# i: @integer;
           t: @text
        enter (i, t)
        do ' ' -> t.put;
           i -> t.putInt
        exit t[]
        #);
   j: @integer;
   s: @text;
   p: @Ptn
do (* first sequence *)
   17 -> j;
   's:' -> s;
   (j,s) -> p;
   p -> putline;

   (* second sequence *)
   17 -> j;
   's:' -> s;
   (j,s) -> &amp;Ptn;
   p -> putline
#)
```

The program consists of a declaration of a pattern: Ptn, and some static items: j, s, and p. The action−part contains two nearly identical sequences of imperatives.

7.6.1 Question A:

Explain the object−structure of this program (i.e. which objects are allocated, and when).

7.6.2 Question B:

Explain the results of executing the first sequences of imperatives.

7.6.3 Question C:

Explain the results of executing the second sequences of imperatives.

7.6.4 Question D:

Explain the difference between the results of the two sequences.

7.7 Assignment

This assignment is a chance to play with the notions of inheritance, specialisation and generalisation. Recall that specialisation involves creating a new subpattern that specialises an existing pattern, that is, it inherits attributes from the existing pattern and (usually) adds new ones of its own. (The subpattern can also extend the superpattern's action part.) Generalisation is when you create a new superpattern by gathering common attributes from several existing patterns.

Starting from the bank example in [Madsen93], write two new subpatterns, CheckingAccount and SavingsAccount, which specialise the Account pattern. Your SavingsAccount pattern should include a new part attribute, interestRate, and a procedure attribute CalculateInterest which can be called at the end of every working day to increment the balance according to the current interest rate.

Your CheckingAccount pattern should model the US system where customers are charged a fixed (small) price for every check they write unless the balance exceeds a certain threshold, in which case writing checks is free. What new part attributes do you need? Add a new procedure attribute ProcessCheck which reduces the current balance by the amount of the check plus the check−writing charge if applicable.

7.8 Assignment

Construct a BETA program that calculates the natural number square root of a number: n >0.

The natural number square root is the number r, such that:

$$r^2 \le n < (r+1)^2$$

Implement this program and test it.

7.9 Assignment

Starting from your own soda−machine code, generalise your SodaMachine pattern by implementing a superpattern called VendingMachine. Also, make another subpattern of VendingMachine called CandyMachine. (A CandyMachine is like a SodaMachine except that instead of juices and colas, you can get chewing gum, chocolate bars, and M&Ms.) Think about which attributes are common to both SodaMachine and CandyMachine; these are candidates for moving up into VendingMachine.

In your original SodaMachine pattern was a procedure attribute that printed the totals, let's call it PrintTotals. Move this procedure attribute up to VendingMachine and specialise it for SodaMachine and CandyMachine. (Don't forget that your definition of PrintTotals in VendingMachine should include an inner imperative.) So SodaMachine and CandyMachine will have new attributes called SMPrintTotals and CMPrintTotals, respectively, which specialise VendingMachine's PrintTotals pattern.

You probably also had a procedure attribute that refilled the machine, let's call it Refill. Make a new Refill attribute in VendingMachine which takes one enter parameter, a number of dollars, and adds this money to the current total. Then specialise Refill in each of SodaMachine and CandyMachine to take extra enter parameters corresponding to the number of items to add (either juices and colas, or chewing gum, chocolate bars, and M&M bags). So SodaMachine and CandyMachine will have new attributes called SMRefill and CMRefill, respectively, which specialise VendingMachine's Refill pattern.

Modify the do−part of your soda−machine code to test your new patterns and attributes.

7.10 Assignment

This assignment is a chance to play with virtuals.

In Assignment 9, you modified your soda machine code to include two new patterns, VendingMachine and CandyMachine. Among the procedure attributes of VendingMachine, were two called Refill and PrintTotals. You also made specialisations of these in SodaMachine and CandyMachine called SMRefill, CMRefill, SMPrintTotals, and CMPrintTotals.

Change these specialised attributes to be virtuals. That is, Refill and PrintTotals should be declared as virtuals in VendingMachine and (the same names) further bound in SodaMachine and CandyMachine. Note that this should be easy! Very little of your other code should need to be

changed.

Add a new virtual called Empty which simply resets to zero the money in the machine and the amounts of all items. (A handy procedure for thieves!) Again, you'll be declaring Empty to be a virtual in VendingMachine and specialising it in SodaMachine and CandyMachine. What part of the work should be done in each pattern?

As usual, be sure to test all your changes using the do−part of the program.

7.11 Assignment

Design a Dictionary pattern with the operations: associate, disassociate, lookup and scan:

- Associate: takes two arguments: entry and element, where element is the lookup−key, and element is the associated element. Associate makes a new association in the dictionary, connecting entry and element.
- Disassociate: takes two arguments: entry and element, and removes their associating in the dictionary.
- Lookup: takes one argument: entry, and returns the associated element.
- Scan: runs the entire dictionary, executing inner for each association. During inner, the two dynamic references entry and element references the entry, respectively element, of the current association.

Discuss the consequences of allowing more than one element for each entry.

Make the dictionary as general as possible, using inheritance and virtual attributes as much as possible.

7.12 Assignment

Modulize the Dictionary pattern from exercise 11 such that the users of the dictionary cannot gain access to the implementation of it.

7.13 Assignment

Extend the Dictionary pattern from exercise 12 such that exceptions are defined taking care of the many different exceptional cases which may occur during the lifetime of a dictionary object.

7.14 Assignment

Make use of the Dictionary from exercise 13 to implement a phone book.

Use the Persistent Store to make it possible to save the phone book between program executions.

Make two distinct programs (e.g. an update and a query program), both using the same phone book saved in the Persistent Store. Its OK to assume that these programs never are running at the same time.

Use the exceptions defined e.g. for dictionary in the phone book and phone book programs to make the programs more fault−tolerant.

7.15 Assignment

Dining philosophers. Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. The room should keep a count of the number of philosophers in it.

The problem is to prevent any philosopher from dying from starvation due to the philosopher on his right always using the right fork. It is assumed, that the philosophers only are eating in a limited period of time.

7.16 Assignment

The following questions require writing a few small programs which draw various geometric figures. The programs should be written in a programming language with the following syntax:

command –> drawN | drawW | drawS | drawE |

moveN | moveW | moveS | moveE |

command newline command |

thickness thickness |

color color

repeat number newline command newline end repeat

thickness –> "thick" | "thin"

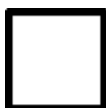
color –> "black" | "red"| "blue"

number –> #an integer greater than 0#

newline –> #a line–feed/carriage–return#

These programs can be realised by implementing a drawing environment using either GuiEnv or Bifrost.

7.16.1 Question A:

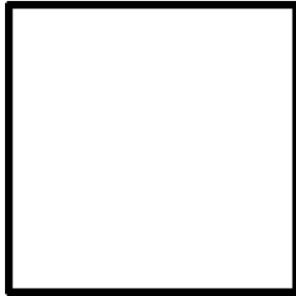


Write and test a program that draws the above figure.

7.16.2 Question B:

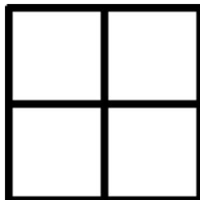
Write and test a program that draws the above figures. Write one or more algorithms that draw the same figures. Are there any essential differences between the algorithms?

7.16.3 Question C:



Write and test a program that draws the above figure. Write one or more algorithms that draw the same figure. Are there any essential differences between the algorithms?

7.16.4 Question D:



Write and test a program that draws the above figure. Write one or more algorithms that draw the same figure. Are there any essential differences between the algorithms?

7.16.5 Question E:

In this assignment, you've used draw–primitives like drawN and moveE. Based on your experience with questions A) through D), suggest other draw primitives. Try writing algorithms with your new primitives. Are yours an improvement? Why?

7.16.6 Tips on using DrawEnv.bet

The BETA implementation of this application can be constructed along the following guidelines. You should have two windows called Drawings and Commands. (There's also a partially hidden Console window which you can ignore.) Choosing an item off the Commands menu adds a line of text with that command (followed by a newline) at the current cursor position in the Commands window. You can also type commands into the window manually. The repeat command menu item works in a slightly special way. If some number of lines in the Commands window are currently selected (i.e. highlighted), then the repeat command will "bracket" those lines. The repeat n part of the repeat statement will be inserted before the selection and the endrepeat after the selection. Be sure to change the number n to be the desired number of iterations. Using the Actions menu, you can reset the current position in the Drawings window back to the upper left corner of the window (for the next draw or move command), clear the contents of both windows, and "run" or "execute"

the current selection in the Commands window. Note: you can copy/paste a saved set of commands into the Commands window from another file (say, a Microsoft word file).

7.17 Assignment

Make a text editor with a menu using MacEnv/AwEnv/MotifEnv. Each time the menu is selected, a beep sound should be made. Compile and run the program.

7.18 Assignment

Write an application with one graphical window, one menu and one dialog box. The menu contains the following entries: Rect, Oval, RoundRect. When one of these are selected, a corresponding shape is drawn in the window. The menu also contains a shape entry. When that entry is selected, a dialog box appears with three radio buttons (Rect, Oval, RoundRect), and four text fields (height, width, X, Y) and two buttons (OK, CANCEL). The radio buttons selects the shape type, height and width defines the size of the shape, and X and Y defines the position of the shape in the window.

Implement the system in MacEnv, AwEnv or MotifEnv and test it.

7.19 Assignment

This assignment is an introduction to MacEnv. Each question asks you to modify one of the small MacEnv demo programs. This require understanding part of the source code for the particular demo program and that in turn means that you'll be referring to the MacEnv documentation . Don't forget to use the index as well as the table of contents when looking up the various MacEnv patterns and attributes.

Notice that for each of the applications, the Close and Quit items on the file menu are automatically enabled. (Close is only enabled if there's a window belonging to the application currently open on the screen.)

7.19.1 Question A: Hammer windows

The Hammer program puts up a window containing a picture of Thor's famous hammer "Mjølner." The window "handles refresh", that is, if the window is activated (clicked in) after having been covered up by another window, it redraws its contents. (Find the part of the code that does this!) The program also allows you to create new hammer windows using the New item on the File menu. Trouble is, these new windows come up in the same place on the screen. Change the code so that the new windows are positioned at different places on the screen, but still have the same dimensions.

7.19.2 Question B: Multiple text editor windows

The program TextEditors lets you create multiple text editor windows using the New item on the File menu. The new windows are positioned successively down and to the right, each one overlapping the previous. Change the program so that new windows are opened next to each other. That is, so they are positioned side–by–side, that is, their edges just touch. (How many new windows can you make before running out of screen space?)

7.19.3 Question C: Catching an event in a text editor window

The EventDemo program creates a text editor window and beeps whenever the mouse is clicked inside the window. Change the program so that it also inserts the text string 'beep' at the current

position. If the user double−clicks in the window, insert the string 'double−beep'. (This gives you a way to experiment with the difference between two successive clicks and a single double−click.)

7.20 Assignment

Make a Matrix data structure with hidden implementation. Make a Vector data structure, such that Vector is able to utilise the implementation of Matrix, without revealing the Matrix implementation to the rest of the program (e.g. to the users of the Vector data structure.

7.21 Assignment

Implement the so−called futures in BETA using the concurrency facilities. A future is a value, returned as the result of an operation, but where the results of the operation may not have been completely calculated yet (e.g. a concurrent computation has not yet completed). References to futures may be passed around like any other object even though the result still isn't available. If some system object evaluates the future, it should be blocked until the result of the computation becomes available, in which case the result(s) are exited from the future object.

8 Course Projects

8.1 Topics

- Project I: Subways in Århus
- Project II: Simulation Environment
- Project III: Vehicle Registration Office

8.2 Project I: Subways in Århus

What?? Subways in her form of public transportation to a system that's already working fine? And by the way, has anyone talked to the potential users about this crackpot idea?

Imagine that it's 15 years hence and the ign of a new subway system. The designers would like to try out various placements of subway stations and lines connecting them – without digging up any streets just yet. Your job is to build a computer–based environment in which they can experiment with locations of subway stations and compare the distances between stations along various routes. You'll be able to reuse bits of your old code (e.g. linked lists and dragable objects), play with a new data structure (the graph), and design "friendly" user interfaces for the city planners.

8.2.1 Modelling a subway system in BETA

A subway station together with its connecting lines is an example of a data structure called a graph. Unlike the tree (see exercise 34), there is no privileged "root" node, nor are there rules requiring that the nodes be organised in a hierarchy (with "parent"/"child" relationships). Rather a general graph is simply a network of nodes and connecting links. You can use a graph to model ng graph nodes model the subway stations and links model the lines that connect stations. So in BETA you could have separate SubwayNode and SubwayLink patterns as well as a SubwayGraph pattern to model the whole system. Then SubwayGraph has two attributes whose values are lists of all the SubwayNodes and all the SubwayLinks, respectively. You'll also want to be sure that each node "knows" about its connecting links and that each link "knows" about the two nodes at its endpoints. You might consider reusing the old LinkedList pattern to represent lists of nodes and links.

8.2.2 Question A: Placing the subway stations

We have to scan a map of enever the user clicks in this window, a new subway station is created with a single character name (starting with A, then B, etc.). A text object is placed in the window at the point of the mouse click displaying that name. If the user subsequently clicks on the name, the station can be dragged to a new location.

Particularly during experimentation with early versions of the system, it might be a good idea to have a menu item which refreshes the window. Another approach would be to implement a first version of this part of the application in a plain window, i.e. with no map on the background.

8.2.3 Question B: The shortest path between two subway stations

Designers of subway systems know that certain routes are more heavily used than others. For example, on a Saturday morning there's always lots of traffic between Gellerup and the market at Ingerslevs Boulevard. If there were stations at those two points, then the subway designer might want to highlight the shortest route between them and be sure that it's direct enough. To support

such experimentation, you need to implement the calculation and display of the shortest path between two stations chosen by the user. The user would select two nodes which are the start and end nodes respectively. Then, with a single menu selection, the shortest route between those nodes is highlighted in the window. (For example, the links along the route could be redrawn with thicker lines.)

How do you compute the shortest path between two nodes in a graph? Here's a textual description of one possible algorithm: The idea is to let each node in the graph calculate the total distance of the shortest route to it from the start node. Then, starting at the end node, you can work your way back to the start node highlighting each link along the shortest path. So there's two steps.

Step 1: Let's first consider how nodes calculate their shortest distance. It's a recursive, object–oriented algorithm. At any given time during the algorithm, each node has a current shortest distance (initially –1 indicating that no calculations have been performed). The start node can immediately improve its current shortest distance to 0 to indicate that the optimal route between a node and itself has 0 length. Now the start node communicates with each of its neighbours along their connecting links. Each neighbour sees whether it can improve on its current value by using the value sent from the start node together with the distance along the connecting link. If this results is a smaller value for the shortest distance to the neighbour node, then it decreases its current shortest distance and starts the same process up with its neighbours. On the other hand, if this does not result in an improved shortest distance, then the neighbour does nothing and does not "pass the baton" along to its neighbours. (Despite the fact that nodes can be visited more than once, you should convince yourself that this recursive algorithm does in fact stop eventually, no matter how complex the graph is.)

Notes:

1. Instead of the actual distance between two points, (which requires computing the square root), you can just use the square of the distance, namely, (y2–y1)^2 + (x2–x1)^2.
2. In addition to remembering the current shortest distance, each node should also remember which incoming link (call it say, PreferredLink) was responsible for that shortest distance.
3. Before running the algorithm you should be sure that any information from previous calculations is deleted. Thus SubwayGraph should have a "cleanup" procedure that runs down all the SubwayNodes and sets all their values to –1 and throws away the reference to the PreferredLink.

Step 2: Now that each node knows the shortest distance from it to the start node and each also knows which incoming link is first along that shortest path (to the start node), it's a simple matter to highlight the shortest path. Just starting at the end node, find the preferred link and highlight it. Then, "pass the baton" (recursively) to the node at the other end of the preferred link. If there is no preferred link, then stop. You should now be at the start node (which will never have a preferred link).

(Note that both of these algorithms are object–oriented and recursive. They are object–oriented because the work is done in procedure attributes that belong locally to each node with the node "handing off" control (i.e. message passing) to the neighbour nodes. They are recursive because these procedures call themselves.)

8.3 Project II: Simulation Environment

This project consists of two parts: Simulation of a group of machines and development of general abstractions for simulation.

8.3.1 Part 1: Simulation of a group of machines

The goal of this part is to construct a model, that makes it possible to simulate a collection of machines, making it possible to evaluate the capacity of the machines in relation to a given flow of production orders.

The system consists of a number of machine groups. Each machine group consists of a number of identical machines. A single machine can handle part of an order.

A production order is characterised by a sequence of machine groups MG1, MG2, ..., MGi and an associated sequence of manufacturing times T1, T2, ..., Ti. The order must therefore be fulfilled by one machine in group MG1 for a time period of T1, followed by a machine in machine group MG2 in T2 period, etc.

New production orders are placed in the system at irregular times. At any given time, the system will have a number of orders under manufacturing. This may cause queues to build at certain machine groups if there is not sufficient manufacturing capacity in the machine group.

The simulation model must simulate arrival and manufacturing of a sequence of production orders for a given period of time. In the simulated period of time, the system must at regular intervals print the number of orders, waiting at each machine group.

8.3.2 Part 2: Development of general abstractions for simulation

As a part of the project, a number of general abstractions (patterns) must be developed. These patterns must enable simulation of discrete events. Describe events is one method for studying the behaviour of large systems of coordinating objects, and for evaluating the effects of changes, that may be too costly to test in practice. The following concepts are characteristic of such systems:

1. A varying number of processes are active simultaneously and gives rise to discrete events at irregular times.
2. Queues may build up when an object must wait to be served by another object, that is serving some other object.

It is not always necessary to deal with real concurrency in the simulation in order to represent processes, that are concurrent in the system we want to make a simulation model for. It is most often sufficient to maintain a sequence of objects with markings of the time of the next time they need to be activated.

An order in the model can e.g. be represented as a process, that wanders from machine to machine, requesting for service. If the machine is idle, the order can immediately be effectuated and the machine can simulate busy for the period of time, it takes to deal with the order. If the machine is busy, the order must wait in a queue of orders, waiting to be serviced by that particular machine. When the machine is finished dealing with an order, it can fetch another order from the queue.

The simulation model will use the concept of simulated time that often is represented by a variable time that denoted the current time and which is updated at certain occasions. The active phases of a process is usually dealt with instantaneously with respect to the simulated time, since it is difficult to simulate a contiguous span of time.

The passing of time is often simulated by the active process suspending execution for some period of time T and the process will then be resumed, when the global time variable have been increased with T.

The general abstraction mechanisms must therefore include:

1. A process concept for simulation of simultaneous processes.
2. A queue mechanism that allows processes to wait. This will imply an operation, e.g.
   wait(Q)
   where Q is a queue and where the result is that the calling process is suspended and entered into the Q. There must also be an operation, e.g.
   activate(X)
   where X is a suspended process, that is removed from the queue, where it is waiting and then resumed.
3. A concept of time. This may be a variable time and a operation , e.g.
   hold(T)
   hat suspends an active process for T time units.
4. An operation to start the simulation, e.g.
   simulate(startProcess, endTime)
   where startProcess is the first active process and endTime is the time, where the simulation is to stop (it is assumed, that the simulation starts at time 0).

8.3.3 Part 3: Development of an interface for simulation

Finally, this project can be extended with demands for constructing an interface, either for the specification of a simulation, or for visualising the dynamics of a running simulation. This interface construction can be realised using GuiEnv or using Bifrost.

8.3.4 Remarks:

It is not expected that this project will result in a full−fledged simulation system. It is part of the project to restrict the project properly and to make the more informal parts of the project formulation precise. The main effort must be places on development and implementation of the general abstractions for simulation.

The design of the system should cover all important aspects of the project. Restrictions must be stated precisely. The implementation should also cover all important aspects. It should be relatively easy to add missing functionality (e.g. without making major changes in the program).

8.4 Project III: Vehicle Registration Office

This project deals with the development of a system for handling the registration of vehicles at the central vehicle registration office.

The register must be able to register different types of vehicles, such as cars, busses, trucks, motorbikes, etc. For each vehicle type, relevant properties such as registration number, owner, weight, cargo, number of passengers, tax, etc. must be registered.

The user of the register must be able to make the following operations:

- Registration of a new vehicle.
- Deregistration of a vehicle
- Change of owner
- Search for vehicles, based on more or less complete informations, such as registration number, owner, etc.
- Printing of statistics for vehicles, such as the distribution of cars by age.

- Printing of receipts for payment of tax and registration of payment of tax.

It is to be expected, that the user may pose additional requirements for functionality, and the design must be prepared of such requests.

The design should argue for the choice of phenomena. The design should also give arguments for the measurable properties and transformations considered for each phenomena. There should also be a discussion of the concepts used for organising the phenomena.

8.4.1 Remarks:

It is not expected that this project will result in a full–fledged vehicle registration system. It is part of the project to restrict the project properly and to make the more informal parts of the project formulation precise.

The design of the system should cover all important aspects of the project. Restrictions must be stated precisely. The implementation should also cover all important aspects. It should be relatively easy to add missing functionality (e.g. without making major changes in the program).

9 Course Materials

The material, used in the different lectures are listed here. This list contains all material mentioned in at least one lecture.

[Borgida85]

A. Borgida: Language Features for Flexible Handling of Exceptions in Information Systems, ACM Transactions on Database Systems, Dec. 1985.

[JLK84,87]

J.L. Knudsen: Exception Handling – A Static Approach, Software Practice and Experience, May 1984.

J.L. Knudsen: Better Exception Handling in Block–Structured Systems, IEEE Software, May 1987.

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

B

Bifrost Graphics System
Block Structure and Part Objects

C

Co–routine Aspects
Compiler and Basic Library
Conceptual Framework for Object–Oriented Programming
Concurrency Aspects
Containers Library
Course Assignments
Course Materials
Course Projects

D

Distributed Objects

E

Event–based programming
Exception Handling
Exceptional Computation and Overview of Language Constructs

F

Freja: the CASE Tool for BETA
Frigg: the User Interface Editor for BETA [2]

I

Imperatives
Introduction to Object–Oriented Programming
Introduction

L

Lectures [2] [3] [4] [5]
Lidskjalv Introduction: Purpose, Overview and
Lidskjalv Patterns

Structure of a Lidskjalv Application

M

Mjølner System Overview
Mjølner System Tools

N

Non–deterministic Aspects

O

Objects, Repetitions, and Patterns
Overview of the Mjølner System

P

Pattern Variables
Persistent Objects
Platform–independent User Interface Construction Using the Mjølner System
Procedural Programming in BETA
Process Communication
Programming–in–the–large in the Mjølner System
Project I: Subways in Århus
Project II: Simulation Environment
Project III: Vehicle Registration Office

S

Sif: the Mjølner Source Browser and Editor
Subpatterns

T

The BETA Programming Language
The Fragment System (Advanced)
The Fragment System (Basics)
The Mjølner System Tools
The Object Browser
The other tools
Topics [2] [3] [4] [5]

V

Valhalla: the Mjølner Source–level Debugger
Virtual Class Patterns
Virtual Procedure Patterns

Y

Ymer: the Mjølner Source Browser and Editor