Teaching ObjecbOriented Programming Using BETA*

Jørgen Lindskov Knudsen
Ole Lehrmann Madsen
Claus Nørgaard[†]
Lars Bak Petersen[†]
Elmer Sørensen Sandvad[†]

Computer Science Department
Aarhus University, Ny Munkegade 116
DK-8000 Aarhus C, Denmark[‡]

April 1990

Introduction

The BETA language is used as a basis in teaching object-oriented programming at Aarhus University. BETA is a modem language that makes it possible to illustrate a large number of concepts in object-oriented programming.

The approach to teaching object-oriented programming is, however, not identical to teaching the BETA language. On the contrary it is emphasized that

* *Presented at the 6th Annual Apple European University Consortium Conference, Salamanca, Spain, April 18th-20th, 1990.*

[†]Address: Mjølner Informatics ApS, Science Park Aarhus, Gustav Wieds Vej 10, DK-8000 Aarhus C, Denmark.

[‡]Personal e-mail addresses: jlknudsen@daimi.dk, olmadsen@daimi.dk, cnorgaard@mjolner.dk, lbpetesen@mjolner.dk, essandvad@mjolner.dk

teaching object-oriented programming is more than teaching object-oriented programming languages. Very often people associate object-oriented progamming with programming in a concrete object-oriented programming language, like Smalltalk. Object-oriented programming and programming in general should not just be a matter of teaching constructs in a given programming language. Most text books on programming languages only describe the *technical differences* between various language constructs. This implies that emphasis is often concentrated around *features* of one language compared to features of another language. This makes it difficult to discuss the *qualitative difference* between languages. The well-known Turing "Tarpit" states that, on theoretical basis, any computation which can be expressed in one of the familiar programming languages can also be expressed in any of the other languages — including Turing machines. This implies that comparison of languages should be more than a discussion about whether or not a given construct may be simulated in another language. Furthermore, a "technical" discussion of programming languages is often lacking arguments about the programmers *perspective* on programming. Instead of technical details, it is often much more fruitful to discuss requirements for supporting one or more perspectives.

Teaching programming languages should concentrate on the conceptual framework underlying the language. For functional programming, it is natural to teach the mathematical framework. For object-oriented programming, the conceptual framework should be discussed. This is not as easy as for functional programming, since the framework for object-oriented programming has not yet been so fully developed as the framework for functional programming.

When teaching object-oriented programming, the conceptual framework for object-oriented programming must of course be supplemented with concrete examples of languages supporting the framework. Here it is important to select a representative set of languages and not just one language. At Aarhus University, BETA is used as one of these languages, representing the class of *strongly typed* object-oriented languages. The reason is that BETA is a modern language that includes most of the constructs of the other strongly typed languages, such as Simula[3], C++[11] and Eiffel[10]. Smalltalk[2] and Scheme[1] are used to illustrate the other class of languages based on dynamic type checking.

This paper consists of three major sections. Section 1 is devoted to a brief discussion of the conceptual approach to teaching programming languages. Section 2 is a brief description of the BETA programming language. Section 3 presents the BETA Macintosh environment.

1 Conceptual Approach to Programming Languages

The approach to teaching programming languages and especially object-oriented programming is very much influenced by the perspective you have on the role of the programming language in the system development process. In fact this role is a three-way role: as a means for expressing concepts and structures (*conceptual modeling*), as a means for *instructing* the computer, and as a means for *managing* the program description. Just focusing on the role as a means for instructing the computer is far to narrow. In the role for conceptual modeling, the focus is on constructs for describing concepts and phenomena. In the role for instructing the computer, the focus is on aspects of the program execution such as storage layout, control flow and persistence. Finally, in the role for managing the program description, focus is on aspects such as visibility, encapsulation, modularity, separate compilation, library facilities, etc.

Some of the success in teaching programming languages can be traced back to the emphasis that is put on using these roles as the foundation of the approach. Here the roles as means for conceptual modeling and prescription have proven very effective, and to some extent this makes the approach to teaching programming languages novel. It has been found that restricting the discussion of programming languages to the role of instruction (or coding) is far to restrictive, primarily because the end-product of a programming process (the program) cannot (and should not) be viewed in isolation from the programming process and thereby the application domain.

A more detailed description of the issues discussed here may be found in [4] and [8], which also contains references to important work that have influenced our research and teaching.

1.1 Programming Perspectives

Teaching the perspective of object-oriented programming cannot (or should not) take place in isolation from other perspectives. Extensive parts of the course should therefore be devoted to programming perspectives as such, and presentation of various different programming perspectives.

Procedural programming[1] is taken as the starting point for the discussion. Functional/logical programming and object-oriented programming are then described as two different reactions to several problems related to the concept of state in procedural programming. In functional/logical programming, the approach has been to eliminate the concept of state, whereas the approach taken in object-oriented programming has been to treat the concept of state as a first-class citizen. In addition various other perspectives such as the process perspective, the type system perspective and the event perspective are treated. The latter three perspectives are not treated extensively but primarily in the context of the other perspectives.

1.2 Object-Oriented Programming

In our approach, object-oriented programming is defined by the following characterization:

> *A program execution is regarded as a* physical *model, simulating the behavior of either a real or imaginary part of the world.*

The object-oriented perspective on programming is in contrast to the above perspectives that are focusing either on manipulations of data structures or on mathematical models. The object-oriented perspective is closer to physics than mathematics. Instead of describing a part of the world by means of mathematical equations, a *physical model* is literally constructed. This means that elements of the program execution are regarded as models of *phenomena* and *concepts* from the real world. Parts of the physical world consists of material. Paper, plastic, wood, Lego bricks are examples of physical material. Objects are the computerized material, used to construct the computer based

[1] To ease the writing, we will use the phrase "...programming" interchangeable with the phrase "the ...programming perspective".

physical models. The part of the world being modeled is described by the program. Some of the well-known examples of languages supporting this perspective are Smalltalk, Beta and C++.

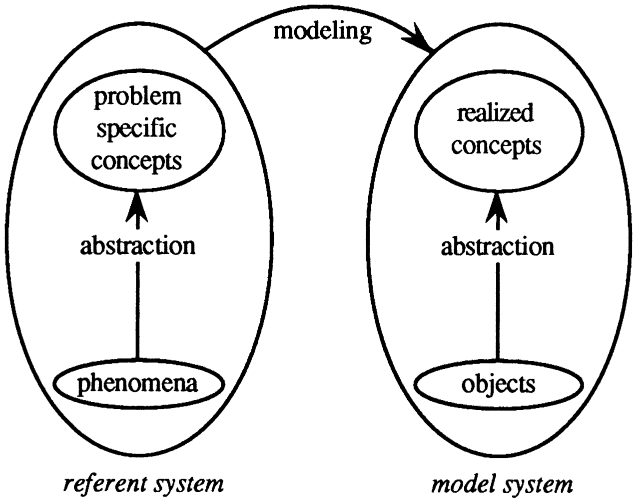
This "definition" cannot be seen in isolation but must be understood in a broader context (this applies for the other perspectives as well.)

1.3 Theoretical Foundation for Object-Oriented Programming

As stated above, the object-oriented perspective must be accessed on basis of a theoretical foundation and not on basis of specific language constructs. The theoretical understanding of object-oriented programming which will be outlined in the following is among others a result of research activities that the authors have carried out together with a number of other people.

Modeling

In order to clarify the different roles that the programming language plays in the prosing process, we have to look more closely at that process. The prosing process may be desks as a modeling process in which several subprocesses take place.



The figure illustrates the programming process as a modeling process between a referent system and a model system. The *referent system* is part of the world that we are focusing on in the programming process, and the *model system* is a program execution modeling a part of the referent system on a computer. The referent system is the concrete physical world or some imagination of a future physical world, and as such it consists only of phenomena. As a characteristic human activity, we create concepts in order to capture the complexity of the world around us — we make abstractions. That is, in the referent system, both phenomena and concepts are important. In the model system, we find elements that model phenomena and concepts from the referent system.
Objects in a Smalltalk program execution are spicily models ofihysical phenomena in the referent system and the sequence of events generated by the execution of a methhod is typically a model of a sub-process going on in the referent system. Concepts in the referent system are modeled by abstractions such as classes, types, procedures and functions. The program text is a *description* of the referent system and in addition it is a *prescription* that may be used to generate the model system.

The programming process can now be described in terms of this figure. During the programming process, three sub-processes are taking place: abstraction in the referent system, abstraction in the model system, and modeling. Please note that intentionally we do not impose any ordering among the sub-processes. *Abstraction in the referent system* is the process where we are perceiving and structuring knowledge about phenomena in the referent system with particular emphasis on the problem domain in question. We say that we are creating *problem specific concepts* in the referent system. This process is an integrated part of the system development process. *Abstraction* *in the model system* is the process where we build structures that should support the model we are intending to create in the computer. We say that we create *realized concepts* in the model system. Finally, *modeling* is the process where we connect the problem specific concepts in the referent system with the realized concepts in the model system.

Concepts

As discussed above, concepts and abstraction are the key notions in our understanding of the programming process. It is therefore necessary to discuss subjects like the notion of concepts and their relations to phenomena, concept understanding, and important aspects of the abstraction process.
A *phenomenon* is something in the world that has definite, individual existence in reality or the mind; anything real in itself. What constitutes a phenomenon is to some degree dependent on the view of the observer. A *concept* is a generalized idea of a collection of phenomena, based on knowledge of common properties of the phenomena in the collection. Concepts may be characterized by three aspects: the designation, extension and intension. The *designation* refers to the collection of names under which the concept is known. The *extension* refers to the collection of phenomena that the concept somehow covers, and the *intension* refers to the collection of properties that in some way characterize the phenomena in the extension of the concept.

These definitions are deliberately somewhat vague since there are (at least two) different ways to understand concepts: the Aristotelian view and the prototypical (or fuzzy) view. Space does not allow an extensive discussion of these two views — just a short characterization. In the *Aristotelian* view, the concepts are rigidly defined, leading to *sharp concept borders* and *relatively homogeneous* phenomena in the extension. The *prototypical* view, on the other hand, is characterized by *blurred concept borders*, phenomena of *varied typicality* in the extension, and *decisionmaking/judgement* when a phenomenon is considered for inclusion in the extension. The prototypical view is the view that best describes human concept understanding.

As it can be seen above, the prosing process is faced with the problem that not only do we restrict the precision of our model by only considering a part of the world (this is a problem studied in system development courses), but equally important, the modeling process *has* to take into account the restrictions imposed by modeling a possible prototypicl concept structure in the referent system into an Aristotelian concept structure in the model system.

Abstraction

In the process of creating concepts it is useful to identify the three well-known sub-processes of abstraction: classification, aggregation and generalization. To *classify* is to form a concept that covers a collection of similar phenomena. To *aggregate* is to form a concept by describing the properties of the phenomena by means of other concepts. And finally, to *generalize* is to form a concept that covers a number of more special concepts based on similarities of the special concepts. All three sub-processes have an inverse process, called *exemplification*, *decomposition* and *specialization*, respectively.

In general the process of creating new concepts cannot just be explained as consisting of the above sub-functions. In practice the definition of concepts wit undergo drastic changes. This is similar to the situation with top-down and bottom-up programming. It is realized by most people that pure top-down or bottom-up development of programs is rarely possible. The understanding obtained during the development process will usually influence previous steps. It is however useful to be aware whether a problem is approached top-down or bottom-up. In the same way it is useful to be aware of the above mentioned sub-functions of abstraction.

The word *abstraction* may be used to characterize a process, and the sub-functions of abstraction were explained as processes going on with the aim of creating concepts. On the other hand the word abstraction may also be used in a static or descriptive way. A concept is an abstraction. Given a number of concepts, their structure may be described in terms of classification, aggregation and generalization. It is e.g. possible to describe a given concept as a generalization of a number of other concepts.

In teaching it is important that the students are aware of this distinction. When evaluating a given language they might consider to what extent the language support abstraction and its sub-functions as a process and to what extent the language supports abstraction and its sub-functions as a means for describing concept structures.

Information Processes and Object-Oriented Programming

Having discussedconcepts and abstraction weturn our attention towards characterizing the part of the world we are interested in creating model systems for, and then characterize object-oriented programming in greater detail.

The kind of model systems we are interested in, are those that model information processes. An *information process* is regarded as a system, developing through transformations of its state. The *substance* or *material* of the process is organized as objects. Objects are the computerized material used to construct computer based physical models. The *state* of the substance may be measured upon through *measurable* properties, and the state of the substance may change as an effect of *transformations* on the substance. Substance is physical material, characterized by a volume and a position in time and space. Substance have certain properties that may be measured. E.g. measurements may be compared with other measurements. Transformations are partially ordered sequences of events that change the substance and thereby its properties. Note that by focusing on information processes, concepts exist that cannot be captured, e.g. “good”, “bad”, etc.

In object-oriented programming, an information process is modeled by organizing the substance of the program execution as a number of *objects*. The measurable properties are modeled as *state* of objects, and transformations are organized as *action sequences* performed by objects. An object is furthermore characterized by a set of *attributes* that may be either *measurable properties*, *part-objects*, *references* to objects, *procedures*, or *classes*. Finally, an object may have an *action-sequence* associated with it. Every object has at any given point in time a state. States are changed by objects performing actions that may involve other objects. Actions may in addition be involved in the production of measurements. A *program execution* consists of a collection of objects. Objects are classified into classes, and classes may be specializations of more general classes.

2 The BETA Programming Language

The rest of this paper will give an overall presentation of BETA. The examples will mainly show how to use BETA to program the Macintosh II[2]. Besides the Macintosh II implementation, implementations exist for SUN-3, HP-90 and Apollo 3500.

This paper is not a complete introduction to BETA. A number of concepts such as co-routine sequencing and concurrency will not be described. For a more detailed description, the reader is referred to [6] and [7].

In the following, it is assumed that the reader is familiar with some other object-oriented language such as Simula, C++, Eiffel or Smalltalk. In [7], object-oriented programming and BETA is presented without assuming previous knowledge of object-orientation.

2.1 Objects and Classes

A BETA program execution consists of a collection of *objects*. Objects are some *computerized material* characterized by a set of *attributes* and an *action-part*. Objects may be described as instances of *patterns*. Consider the following example:

```
Student:
    (# Key:  @ Integer;
       Name:  @ Text;
       Major:  ^ Education;
       CoursesTaken:  @ Set(# elmType::< Course #);

       ChangeMajor:
          (# newMajor:  ^ Education
          enter newMajor[ ]
          do ...
          #) ;
       CoursePassed:
```

[2] The current implementation runs on Macintosh SE/30 and II under MPW and at least 4 Mb of memory.

```
            (# C: ^ Course
            enter C[ ]
            do C[ ] -> CoursesTaken.Insert
            #)
         CourseGrade:
            (# C: ^ Course;
            G: @ Grade
         enter C[ ]
         do (if (C[ ] -> CoursesTaken.has)\\
            TRUE then C.grade -> G if)
         exit G
            #)
      #)
```

The example describes:

- A pattern with the *name* `Student`.
- The *structure* of the pattern is described by `(# ...#)`.
- An instance of `Student` represents a student enrolled at some university.
- Student objects are characterized by seven attributes: `Key, Name, Major, CoursesTaken, ChangeMajor, CoursePassed` and `CourseGrade`.
- The attributes `Key`, `Name` and `Major` and `CoursesTaken` are *reference attributes*. The distinction between the meaning of `@` and `^` will be explained below.
- The attribute `Key` is a unique key associated with all students.
- The attribute `Name` represents the name of the student.
- The attribute `Major` refers to an object describing the major follows by the student.
- The attribute `CoursesTaken` refers to an object that keeps track of the courses that the student has completed. The construct `Set (# elmType ::< Course #)` describes that `CoursesTaken` is an instance

of the pattern `Set`. `Set` is a generic pattern parameterized by a pattern `elmType`. The "pattern parameter" is bound to the pattern `Course`. This will be further explained below.

- The attributes `ChangeMajor`, `CoursePassed` and `CourseGrade` are pattern attributes.
- An instance of `ChangeMajor` represents an action-sequence to be executed when the student changes to another major.
- An instance of `CoursePassed` represents an action-sequence to be executed when the student completes a course.
- An instance of `CourseGrade` represents an action-sequence to be executed in order to find the grade a student have obtained in a course.

The pattern `Student` may be used to describe `Student` objects as follows:

```
S1, S2, S3:  @ Student
```

Using `S1`, `S2`, and `S3`, it is possible to access attributes of the student objects:

```
S1.Name
```

denotes the `Name` attribute of the `Student` object referred to by `S1`. The name of the `Student` object may be changed by means of an *assignment imperative* of the following form:

```
'Hans Christian Andersen' -> S1.Name
```

Description of an action sequence representing that a course has been taken by student `S1`, may be described as follows:

```
advancedPhysics[ ] -> &S1.CoursePassed
```

This describes that an instance of `S1`'s `Coursepassed` attribute is generated. A reference to the object `advancedPhysics` is assigned to the `C` attribute of this new object (described by `enter C[ ]`). The imperatives described in the `do`-part of `CoursePassed` will then be executed. The symbol `&` reads `new`. This coesponds to message sending in Smalltalk and remote procedure call in Simula , C++ and Eiffel. In this case, where `CoursePassed` is used as a procedure pattern, we could have used the shorthand:

```
advancedPhysics[ ] -> S1.CoursePassed
```

A *pattern* is an abstraction mechanism intended for representing concepts in general. Very often a pattern is intended for modelling either a class, procedure or function. The pattern construct is a generalization of abstraction mechanisms such as class, procedure, and function. A pattern may consequently be used as a class, procedure or function. A pattern that is intended to be used as a class will be called a *class pattern*. Similarly we shall speak about *procedure patterns* and *function patterns*. In the student example, `Student` corresponds to a class pattern and `ChangeMajor` and `CoursePassed` correspond to procedure patterns.

2.2 Part Objects and Separate Objects

The attributes `Key`, `Name`, `Major` and `CoursesTaken` of the class pattern `Student` are examples of *reference attributes*. Reference attributes corresponds to instance variables in Smalltalk and member fields in C++. A reference denotes an object. A reference may either be *static* or *dynamic*. A static reference will constantly denote the same object whereas a dynamic reference may denote different objects. A static reference is described in the following way:

```
S: @ Student
```

where `S` is the name of the reference and `Student` is a pattern name. The object being denote is generated as part of the generation of the enclosing object. A static reference denotes the same object during the lifetime of the enclosing object. An object generated in this way is called a *part object*.

A dynamic reference is described in the following way:

```
R: ^ Student
```

where `R` is the name of the reference. A dynamic reference may denote different objects during the lifetime of the enclosing object. Initially it denotes `NONE`, which represents no object. A dynamic reference may be given a value by means of a reference assignment like:

```
S[ ] -> R[ ]
```

which describes that a reference to the object referred to by `S` is assigned to `R`. This means that `R` and `S` will both refer to the same object after the assignment. In this example, `S` is a static reference referring to a part object. A dynamic reference may of course also be assigned to `R`. If

```
R1:  ^ Student
```

then

```
R1[ ] -> R[ ]
```

is another example of a reference assignment. Note that an assignment of the form

```
R[ ] -> S[ ]
```

is not legal, since `S` is a static reference.

It is also possible to create objects dynamically by execution of actions. The following evaluation creates an instance of the pattern `Student` and the result of the evaluation is a reference to the newly created object:

```
& Student[ ]
```

As for procedure invocation, the symbol `&` means `new`. The symbol `[ ]` means that a reference to the object is returned as the result of the evaluation. A dynamic generation may be part of a reference assignment:

```
& Student[ ] -> R[ ]
```

The result of this assignment is that a new instance of `Student` is created and a reference to this new object is assigned to `R`.

2.3 Subpatterns

Patterns may be organized in a subpattem hierarchy. The following example shows a subpattem hierarchy of `Record`, `Person`, `Employee`, `Student`, and `Book`.

```
Record:
    (# Key:  @ Integer;
    #);
Person:   Record
    (# Name:  @ Text; Sex:  @ Text
    #);
Employee:  Person
    (# Salary:  @ Integer; Position:  @ Text;
    #);
Student:  Person
    (# Major:  ^ Education;
       CoursesTaken:  @ Set(# elmType::< Course #)
    #);
Book:  Record
    (# Author:  ^ Person; Title:  ^ Text
    #)
```

This example is of course not complete. Only attributes particular relevant to the following examples are included. Please note, that the `Student` pattern described earlier is introduced as a subpattern in this hierarchy. All attributes of the previous definition should of cource be repeated here.

`Person` is defined as a subpattem of `Record`, which means that it inherits the description of `Record`. `Student` and `Employee` are both defined as subpatterns of `Person`. Finally `Book` is also defined as a subpattem of `Record`. The notion of *superpattern* is the reverse of subpattern. `Record` is the superpattern of `Person` and `Book`. `Person` is the superpattern of `Student` and `Employee`. The meaning of subpatterns is similar to subclassing in Simula, Smalltalk and most other object-oriented languages. An instance of a subpattem has all the attributes of the superpattern in addition to the new attributes described for the subpattem.

2.4 Qualification of References

In BETA, references are qualified as in Simula, C++ and Eiffel. The qualification of a reference restricts the set of objects that may be referred to by the reference. Consider refe nces declared as follows:

```
R: ^ Record;
P: ^ Person;
S: ^ Student
```

The qualification of `R` is `Record`, the qualification of `P` is `Person`, and the qualification of `S` is `Student`. `R` may refer to instances of `Record` or instances of subpatterns of `Record`, `P` may refer to instances of `Person` or instances of subpattems of `Person`, and `S` may refer to instances of `Student` or instances of subpattems of `Student`. Intuitively, we say that `R` must be at least a `Record`, `P` at least a `Person`, and `S` at least a `Student`.

2.5 Values and Assignments

One of the fundamental concepts inprogramming is the distinction between the reference to an object and the state of the object. Many object-oriented languages does not express this difference explicitly. In the BETA syntax there is an explicit distinction between manipulating a reference and manipulation of the state of an object.

Consider the following pattern:

```
Point:
    (# x, y:  @ Integer; count:  @ Integer;
    enter (x, y)
    do count + 1 -> count
    exit (x, y)
#)
```

Point objects can be manipulated by manipulating references and by manipulating the state. The two different assignments are called value and reference assignments. The `enter` part specifies, that two values may be assigned to a `Point` object, and the `exit` part specifies that two values may be extracted from a `Point` object. Assignment to `Point` objects will result in the manipulation of the `x` and `y` part objects of the `Point` objects, whereas the `count` part objects is not affected. On the other hand, `count` cannot be extracted from a `Point` object by an value assignment to another `Point` object.

Consider the following object:

```
(# P1, P2:  @ Point;
   P3, P4:  ^ Point
do &Point[ ] -> P3[ ];  &Point[ ] -> P4[ ];
   (1, 1) -> P1;  (2, 2) -> P2;             {1}
   (3, 3) -> P3; (4, 4) -> P4;              {2}
   ...                                      {3}
   P1 -> P2;  P1 -> P3;  P3 -> P4;          {4}
   P1[ ] -> P3[ ];  P3[ ] -> P4[ ];         {5}
#)
```

At {1}, `P1.x` and `P1.y` are both `1`,and `P2.x` and `P2.y` are both `2` where as `count` in both `P1` and `P2` is 0 (initial value of `Integer`). At {2}, `P3.x` is `3`, etc. All assignments are value assignments, transferring values into the objects through the `enter` list. Let us assume, that ... at {3} results in `P1.count` being `1`, `P2.count` being `2`, etc., whereas the `x` and `y` values are unchanged. At {4}, the assignments are still value assignments. This implies that `P3.x` is `1` but `P3.count` is still `3` (since `count` is unaffected by value assignments since it is not in the `enter/exit` lists). However, the assignments in {5} are reference assignments resulting in `P1`, `P3` and `P4` pointing at the same object, and the `count` attribute of this object is unaffected by these assignments and thus still being `1`.

The legality of a value assignment is tested by matching the `exit` list with the `enter` list. An `exit` list matches an `enter` list, if the two lists have the same (recursive) structure, and if two identical basic values (such as `Integers`) are matched, or two dynamic references are matched. A dynamic reference in the `exit` list (`A`) matches a dynamic reference in the `enter` list (`B`), if the qualification of `A` is a subpattem of the qualification of `B`.

The legality of a reference assignment (`A[ ] -> B[ ]`) is tested by checking that `B` is a dynamic reference, and that the qualification of `A` is a subpattem of the qualification of `B`.

2.6 Combination of Action Parts

Since patterns may be used as procedures, the subpattem mechanism is also available for procedure patterns. Execution of a subpattem consists of executing the `do`-part of the superpattern combined with the `do`-part of the subpattem. The combination of `do`-parts is described by means of the `INNER` imperative. Assume that `PP` is a subpattern of `P`. Execution of an instance of `PP` starts by an execution of the `do`-part of `P`. Whenever an `INNER` is encounted in the `do`-part of `P`, the `do`-part of `PP` gets executed.

The following example illustrates the combination of `do`-parts of subpattems. We consider three levels of subpattems:

```
OpenRecord:
    (# ID: ^ Text; R: ^ Record
    enter ID[ ]
    do ID[ ] -> theDataBase.Open -> R[ ];
       INNER;
       R.Close
    #);

OpenWritableRecord:  OpenRecord
    (#
    do R.Lock;
```

```
        INNER;
        R.Free
    #);

NewKey:  OpenWritableRecord
    (# newKey:  @Integer
    enter newKey
    do newKey -> R.Key
        INNER;
    #)
```

The above exarnple describes a hierarchy of procedure patterns. `OpenWritableRecord` is a subpattem of `OpenRecord` and `NewKey` is a subpattem of `OpenWritableRecord`. The pattern `NewKey` may be invoked in the following way:

```
(R[ ], 1234) -> &NewKey
```

The meaning of this is as follows:

- An instance of `NewKey` is created
- The enter part of this instance is a concatenation of the enter parts of `OpenRecord`, `OpenWritableRecord` and `NewKey`. This gives the following enter part:

  ```
  enter(ID[ ], newKey).
  ```

- Execution of the `NewKey` object starts with the execution of the `do`-part of the topmost superpattem of `NewKey`. This means that execution of the `do`-part of `OpenRecord` is executed. menever `INNER` is executed here, the `do`-part of `OpenWritableRecord` is executed. Whenever `INNER` is executed here, the `do`-part of `NewKey` is executed. Execution of `INNER` in the `do`-part of `NewKey` is the empty action (`skip`), since `NewKey` is the procedure pattern being invoked.
- Execution of `NewKey` gives rise to execution of the following actions:

```
ID[ ] -> theDataBase.Open -> R[ ];
    R.Lock;
       newKey -> R.Key;
           skip;
    R.Free;
R.Close
```

2.7 Virtual Patterns

In Simula and C++, a procedure attribute may be declared as a virtual procedure attribute. In BETA, a pattern attribute may be declared virtual. A virtual pattern attribute may be extended in subpattems. Here the virtual concept of BETA will be illustrated by means of examples. For a more detain description, see [9]. In the following example, a `Display` attribute has been added to the `Record` pattern hierarchy. The procedure pattern `Display` is declared as a visual pattern. The description of `Display` is then extended in the subpatterns `Person` and `Student`.

```
Record:
    (#  Key:  @ Integer;
       Display:< (# do {Display Key} INNER #)
    #)
Person:  Record
    (#
       Display::< (# do {Display Name, Sex} INNER #)
    (#
Student:  Person
    (#
       Display::< (# do {Display Major, CoursesTaken}
           INNER #)
    #)
```

The construct

```
Display :< (# ...#)
```

in `Record` is a declaration of a virtual pattern attribute called `Display`. For a virtual pattern only part of its structure has been described. This is different from a non-virtual pattern where the complete structure is described as part of the declaration. The construct

```
Display ::< (# ...#)
```

in `Person` and `Student` describes that the structure of the `Display` pattern is extended.

Consider the reference

```
R: ^  Record
```

and the invocation

```
R.Display
```

If `R` refers to an instance of the pattern `Record`, then the `Display` pattern described in `Record` will be invoked. If `R` refers to an instance of `Person`, the `Display` pattern of `Person` will be invoked and similarly if `R` refers to an instance of `Student`, the `Display` pattern of `Student` will be invoked.

It has not yet been said exactly what it means to invoke the `Display` pattern of, say `Student`. There is an important difference between the virtual procedure mechanism of Simula, C++ and Eiffel and the virtual (procedure) pattern mechanism of BETA. In Simula, C++ and Eiffel, the definition of a virtual procedure is completely redefined in subpatterns. For C++ and Eiffel, it is possible explicitly to call the virtual procedure of the superpattern.In BETA, it is not possible to redefine a virtual pattern. It is "only" possible to extend the definition of a virtual pattern.

In the above example, the `Display` pattern of `Person` is a subpattem of the `Display` pattern of `Record`. Similarly, the `Display` pattern of `Student` is a subpattem of the `Display` pattern of `Person`. In fact, anonymous patterns corresponding to

```
Record-Display:  (# do {Display Key} INNER #);
Person-Display:
    Record-Display (# do {Display Name and Sex} INNER #);
Student-Display:
    Person-Display(# do {Display Major and CoursesTaken}
                         INNER #);
```

are created as a result of the virtual definitions and extensions hereof.

For instances of `Record`, `Display` is bound to `Record-Display`, for instances of `Person`, `Display` is bound to `Person-Display` and for instances of `Student`, `Display` is bound to `Student-Display`.

Let us reconsider an invocation of

```
R.Display
```

Assume that `R` refers to an instance of `Student`. This implies that `Student-Display` will be invoked and that the following actions are executed:

```
{Display Key}
    {Display Name and Sex)}
       {Display Major and CoursesTaken}
```

It is possible to use explicit qualification of virtual patterns instead of the anonymous patterns mentioned above. This means that we could describe the `Record` hierarchy as follows:

```
Record:
    (# Key:  @ Integer
       Display:< DisplayRecord;
       DisplayRecord:  (# do {Display Key} INNER #)
    #);

Person:  Record
       Display:< DisplayPerson;
```

```
        DisplayPerson:
            DisplayRecord:  (# do {Display Name and sex}
                               INNER #)
    #);

Student:  Person
    (#
        Display::< DisplayStudent;
        DisplayPerson:
            DisplayPerson:  (# do {Display Major
                               and CoursesTaken} INNER #)
    #)
```

2.8 Virtual Class Patterns

The above examples of virtual patternsis an example of using the virtual mechanism as virtual procedure patterns. Since the pattern is a unification of class, procedure and function, the virtual mechanism is also available for classes. This means that it is possible to describe virtual class patterns. A virtual class pattern is declared in the same way as a virtual procedure pattern. Consider the following description of the pattern `Set`:

```
Set:
    (# elmType :< Object; {``Type'' of elements in the Set}
        Insert:  (# E: ^ elmType enter E[ ] do ...#);
        Has:
            (# E: ^ elmType; found:  @Boolean
            enter E[ ]
            do ...
            exit found
            #) ;
        Remove:  (# E: ^ elmType enter E[ ] do ... #);

    Scan:
        (#      current:  ^ linkage;
                thisElm:  ^ elmType;
```

```
            atEnd:  @Boolean
    do head[ ] -> current[ ];
        Loop:
            (if (current[ ] <> NONE) / / True then
            current.elm[ ] -> thisElm[ ];
            (current.succ[ ] = NONE) -> atEnd;
            INNER;
            current.succ[ ] -> current [ ];
            restart Loop
    if) #);
    {Implementation attributes}
    Linkage:  (# E: ^ elmType; succ:  ^ linkage #);
    head:  ^ Linkage;
#)
```

A `Set` object is characterized by the procedurepatterns `Insert`, `Has`, `Remove` and `Scan`. The procedure pattern `Insert` inserts an object into the `Set`, `Has` test whether or not a given object is in the set, and `Remove` removes a given object from the set. The set is implemented as a linked list. The class pattern `Linkage` describes the objects of such a list. Each object in the list has a reference, `E`, to the object in the set. In addition, it has a reference, `succ`, to the next element of the list. The reference `head` refers to the first element of the list. The details of the procedure patterns `Insert`, `Has` and `Delete` are not described.

The attribute `Scan` is an example of a *control pattern*. An execution of `Scan` goes through all the elements of the set and perform an `INNER` for each element of the set. The reference `thisElm` functions as an index variable that steps through the elements of the set. An exmple of using `Scan` is shown below.

The virtual pattern `elmType` is the “type” of the elements of the `Set`. `ElmType` is qualified by the most general pattern `Object`. This means that a `Set` object may include instances of all patterns. In subpattems of `Set`, it is possible to restrict the type of elements to be stored in the set. This may be done by extending the virtual pattern `elmType` to subpattems of `Object`. The following example shows an example of a set for storing `Records`. The class pattern `RecordSet` is defined as a subpattem of `Set`. The virtual pattern `elmType` is extended to the pattern `Record`. This restricts the elements in `RecordSet` to instances of `Record` (or subpattems of `Record`).

```
RecordSet:  Set
    (# elmType ::< Record;
       Display:  (# do scan(# do thisElm.Display #) #)
    #)
```

A procedure pattern attribute `Display` has been added to `RecordSet`. It scans through all elements of the `RecordSet` and invokes their `Display` pattern. This is legal since `thisElm` is quality as `Record` in `RecordSet` and `Record` is known to have a `Display` attribute.

It is possible to describe further subpattems of `RecordSet`:

```
PersonSet:  RecordSet(# elmType::< Person #);
StudentSet:  PersonSet(# elmType::< Student #);
```

A `PersonSet` may contain instances of `Person` and instances of subpattems of `Person` whereas `StudentSet` may contain instances of `Student` and instances of subpattems of `Student`.

The attribute `CoursesTaken` described in pattern `Student` is another example of using pattern `Set`. The description of this attribute is as follows:

```
CoursesTaken:  @ Set(# elmType::< Course #)
```

This is an example of a *singular object.* The object `CoursesTaken` is described directly and not as instance of a pattern. Instead, it is possible to introduce a new pattern for this purpose:

```
CourseSet:  Set(# elmType::< Course #);
CoursesTaken:  @ CourseSet
```

In general, a singular object description may introduce an arbitrary number of new attributes. It is not just restricted to have virtual bindings as in the above example.

The above examples have shown examples of nested patterns. The `Set` pattern contains local patterns like `Insert` and `Linkage`. In general, BETA supports *block structure* in the sense that patterns may be arbitrarily nested. In addition to nesting procedure patterns, it is also possible to have nested class patterns as demonstrated above with `Set` and `Linkage`.

3 The BETA Macintosh Environment

The BETA implementation for the Macintosh II consists of an implementation of the BETA language with automatic garbage collection and an extensive set of libraries, containing patterns describing often used data structures and a complete interface to the Macintosh Toolbox (the BETA Macintosh interface is called `MacEnv`). Besides the Toolbox interface, `MacEnv` contains an object-oriented layer with patterns describing windows (with and without graphics capabilities), menus (pull-down and pop-up), dialog boxes and text editors. Moreover there is an object-oriented graphics library. The event handling within interaction objects (such as windows, menus, etc.) are handled by means of virtuals. Furthermore, `MacEnv` contains object-oriented interfaces to the mouse, the cursor, the clipboard, the active window and the menu bar of the Macintosh.

The complete Mjølner BETA system includes a syntax-directed editor, a metaprogramming system, a fragment system, a user interface system, and several additional libraries. The system is available on SUN-3, Apollo 3500, HP 9000 and Macintosh II and SE/30. An overview of the system is given in [5].

The BETA Macintosh interface has the form of an *abstract superpattern* `MacEnv` and applications are created by making specializations of this pattern. By an abstract superpattern is meant that the pattern has to be specialized in order to supply additional information (patterns, extensions to virtual patterns, references, etc.) to complement the entire application.

In the following, parts of `MacEnv` will be described by giving an example of using `MacEnv`. `MacEnv` has been chosen because it is well suited for demonstrating the features of the BETA programming language. In addition it illustrates that by using BETA on the Macintosh, it is easy for students to program their own applications using the powerful Macintosh Toolbox.

Let us present the facilities of `MacEnv` by creating a Macintosh application with an interface to the `Record` example presented above. The application will present itself by a window showing the inheritance graph of `Record`. Each node in the graph presents a pattern and the leaves in the graph represents both the pattern and an instance of `Register`, containing instances of that pattern. Users may interact with each node by pressing the mouse button on each node, resulting in a pop-up menu, showing the valid commands of that node. Figure 1 shows the graph with the user pressing the mouse button on the `Book` node. Three menu items are available: `Greate`, `Find` and `PrintAll`. `Create` opens a dialog box containing a template for a `Book` object to be filled in by the user. When the `OK` button is pressed, a `Book` object is created and inserted in the `Book` register.

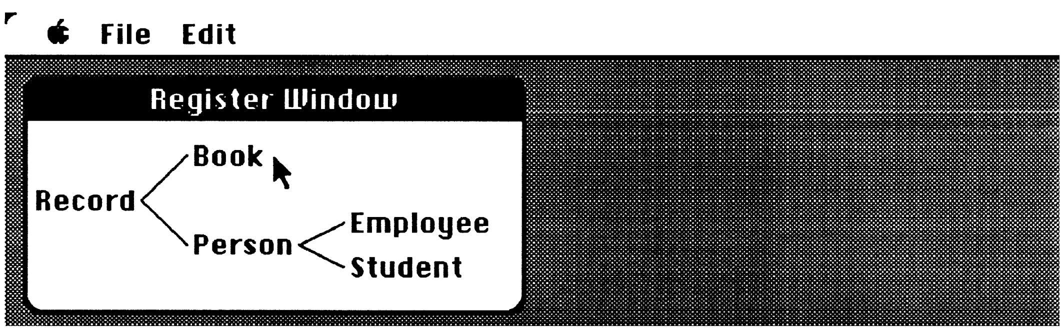


Figure 1: The inheritance graph window

The `Find` menu item opens a dialog box. Here only selected fields needs to be filled in, and when the `Next` button is pressed, the first object in the `Book` register which matches the selection pattern will be shown (i.e. contains identical values in the fields filled in by the user). This dialog box has two buttons: `Next` which displays the next object in the `Book` register and `Ok` which closes the dialog box.
The `PrintAll` menu item will display all objects in the `Book` register in separate windows.

Similar menus are available on the `Employee` and `Student` nodes. On the `Person` node, the `Create` menu item is not available and the `Find` and `PrintAll` items will work on both the `Employee` and `Student` registers. On the `Record` node, `Find` and `PrintAll` are available too, but `Find` will only be able to select on a given `Key` value.

In the following program, realizing this interface, all dialog axes and menus are descry using Macintosh resources. This is done in order to reduce the size of the example.

3.1 Creating the Top Level Window

In order to indicate which parts of the following is inherited from `MacEnv` (and possibly further extended), we have written all names etc. originated from patterns defmed in `MacEnv` in *italic*.

```
Macenv
(#
 (* Record example as above *)
 GraphWindow: @graphicsWindow
   (#
     recordNode: @hitTextObject(# ...#);
     personNode: @hitTextObject(# ...#);
     employeeNode: @hitTextObject(# ...#);
     studentNode: @hitTextObject(# ...#);
     bookNode: @hitTextObject(# ...#);

     EventHandler::<
       (#
         refresh::<
           (# do
             (70,15) -> pen.moveTo; (50,35) -> pen.lineTo;
             (70,55) -> pen.moveTo; (50,35) -> pen.lineTo;
             (140,65) -> pen.moveTo; (120,55) -> pen.lineTo;
             (140,45) -> pen.moveTo; (120,55) -> pen.lineTo
           #)
       #);

         Open::<
           (# do
             (true, 'Record', fonts.system, 12, (3,40)) ->
```

```
                recordNode.init;
              (true, 'Book', fonts.system, 12, (73,20)) ->
                bookNode. init;
              (true, 'Person', fonts.system, 12, (73,60)) ->
                 personNode. init;
              (true, 'Employee', fonts.system, 12, (143,50)) ->
                 employeeNode. init;
              (true, 'Student', fonts.system, 12, (143,70)) ->
                 studentNode. init;
          #)
  #)
do (* MacEnv *)
  'Register' -> GraphWindow.open;
#)
```

The top level window, `GraphWindow`, is a singular instance of `graphicsWindow`. In `GraphWindow` and five singular instances of `hitTextObject` are defined: `recordNode`, `personNode`, `employeeNode`, `studentNode` and `bookNode`. Furthermore, the virtual procedure pattern `EventHandler`, inherited from `graphicsWindow`, is extended to draw the connecting lines in the hierarchy in response of a `refresh` event. The procedure pattern `open`, also inherited from `graphicsWindow`, is specialized to initialize the `hitTextObjects`. `HitTextObjects` are textual objects, that are selectable with the mouse. The parameters to the initializations specifies that the objects are selectable, the text of the object, the font to be used, the size of the font, and finally the position to draw the object in the window. `HitTextObjects` are automatically redrawn in response of `refresh` events. Finally, the only action of the body of `MacEnv` is to initialize `GraphWindow`. `'Register'` is the name of the Macintosh resource defining the window type. `Open` fetches the resource, initializes the window and displays the window.

We have not defined what happens when the user selects one of the nodes. This is done by defining menus and dialog boxes and associate them with the user actions of selection. This is done in the `hitTextObjects`.

3.2 Creating Menus

For each node, we have to define the menus. We will only show the definition of the `bookNode` in fur details here:

```
bookNode: @hitTextObject
    (# bookRegister: @BookSet;
       bookMenu: @menu;

       CreateDialog: dialog(# ...#);
       findDialog: dialog(# ...#);

       hit ::<
          (# do (if ((mouse.getPosition, 0) ->
                        bookMenu.Popup)
                //1 then CreateDialog
                //2 then findDialog
                //3 then bookRegister.Display
                if)
          #) ;
       init ::<
          (# do
                bookRegister.init;
                'Book' -> bookMenu.namedGet;
                bookMenu -> MenuBar.InsertSubMenu;
                    (* register as a pop-up menu *)
          #) ;
    #)
```

`BookNode` is being defined as a singular instance of `hitTextObject`. `BookNode` contains the `bookRegister`, a menu and two dialog boxes. The `bookNode` extends the inherited `hit`, which defines the action to be executed when the book label in the inheritance tree is invoked. In this case, the actions pops up a menu with three choices: `Create`, `Find` and `PrintAll` (defined by the `'Book'` resource). `Mouse.getPosition` returns the current position of the mouse. The numbers `1`, `2` and `3` are the numbers of the menu choices. In the case of the first menu choice being selected (i.e. `Create`), the `createDialog` box is opened. In the case of the second menu choice being selected (i.e. `Find`), the `findDialog` box is opened. Finally, in the case of the third menu choice being selected (i.e. `PrintAll`) , the whole `bookRegister` will be printed. The inherited initialization pattern, *init*, initializes `bookRegister`, fetches the `'Book'` resource that defines the format of `bookMenu`. Finally, `bookMenu` is inserted in the menu bar in order to enable it to be used as a pop-up menu. The inherited procedure pattern `namedGet` fetches a resource with the given name, and initializes the object according to the specifications in the resource. `NamedGet` is also used in the definition of the dialog boxes below. The details of the flog boxes are given in the next section.

3.3 Creating Dialog Boxes

The dialog boxes are all created as singular instances of the `Dialog` pattern:

```
createDialog:  @dialog
    (# EventHandler ::<
        (# itemSelected ::<
            (# do
                (if item
                //1 then (*OK button*)
                    (newKey, 4-> items.getText,
                        6-> items.getText) -> newBook ->
                    bookRegister.insert;
                    true -> terminated
                //2 then (*Cancel button*)
                    true -> terminated
        if) #) #)
    do
        'CreateBook' -> namedGet
    #)
```

The `createDialog` box contains two buttons (defined by the `'CreateBook'` resource), and the inherited `EventHandler` is extended to specify that a new instance of `Book` should be initialized and inserted into into the `bookRegister`, based of the information entered in the various welds of the dialog box, if the OK button is pressed. The unique Key field of the new Book instance is created by the newKey procedure pattern (not shown). If the Cancel button is pressed, the dialog box will be closed without inserting any new Book objects into the bookRegister. The numbers 4 and 6 refers to the text fields of the dialog box. A dialog box is closed by invoking the *terminated* procedure pattern with true as enter value.

FindDialog is a little more complex than createDialog since it must enable the scanning of bookRegister and show each book that has the same value for the specified fields. This is implemented by means of a co-routine[3] Scanner that returns a matching Book instance in each invocation. The details of Scanner is deferred until after the details of findDialog.

```
findDialog: dialog
    (# moreBooks:  @Boolean;
       Scanner:  @|bookRegister.Scan(# ...#);

       EventHandler ::<
           (# itemselected ::<
               (# do
                   (if item
                   //1 then (*OK button*)
                       true -> terminated
                   //2 then (*Next button*)
                       (if moreBooks//TRUE then
                           (5 -> items.getText,
                            7 -> items.getText ) -> Scanner
                              -> moreBooks
                       if)
           if) #) #);

    do
       'FindBook' -> namedGet;
       true -> moreBooks;
```

[3]BETA co-routines are not described above, but for the sake of this example it is sufficient to think of them as procedures, that can be suspended in the middle of their execution. Next time, the co-routine is invoked, it will resume from the suspension point. Each time a co-routine is suspended, it will return the values specified in the exit-list.

```
    (8, ' ') -> items.setText;
    (10, ' ') -> items.setText;
    (11, ' ') -> items.setText;
#);
```

The `findDialog` box contains two buttons (defined by the `'FindBook'` resource), and the inherited *EventHandler* is extended to specify that the selection patterns, entered by the user in the text fields 5 and 7 in the dialog box, are given as enter parameters to `Scanner` when the `Next` button is pressed. If the `OK` button is pressed, the dialog box is closed. The `do`-part of `findDialog` specifies fetching the `'FindBook'` resource, and clearing the text fields 8, 10 and 11 in the dialog box (the text field numbers are defined in the resource). The boolean variable `moreBooks` is used to test, whether `Scanner` is able to deliver any more books. The details of `Scanner` are given below.

```
Scanner: @|bookRegister.Scan
    (#title, author: @text;
      matching:
        (# match: @Boolean
        do true -> match;
          (if (title.length > 0)//TRUE then
            (title -> thisElm.title.equal) -> match
          if);
          (if (match and (author.length > 0))//TRUE then
            (author -> thisElm.author.equal) -> match
          if)
      exit match
      #) enter (author, title)
    do
        (if matching//TRUE then
          (9, (thisElm.Key -> Integer2Text)) -> items.setText;
          (11, thisElm.author) -> items.setText;
          (13, thisElm.title) -> items.setText;
          SUSPEND
        if);
      exit atEnd
    #);
```

Scanner is a specialization of bookRegister.scan, which given an author and a title (possibly empty text strings), scans through the register, and checks whether the book in question has identical values in the author and title fields (if the corresponding enter value is non-empty). If such a book is found (i.e. the matching procedure pattern returns true), the text fields 9, 11 and 13 are initializes to show the values of the book found in bookRegister. The procedure pattern Integer2Text converts a text containing a integer literal, into the corresponding integer (not shown). Hereafter, Scanner is suspended, waiting for the user to press the next button. Scanner returns a boolean (atEnd defined in scan) indicating whether is can be resumed in order to display another book.

Figure 3 shows a screen dump of the three dialog boxes described above.

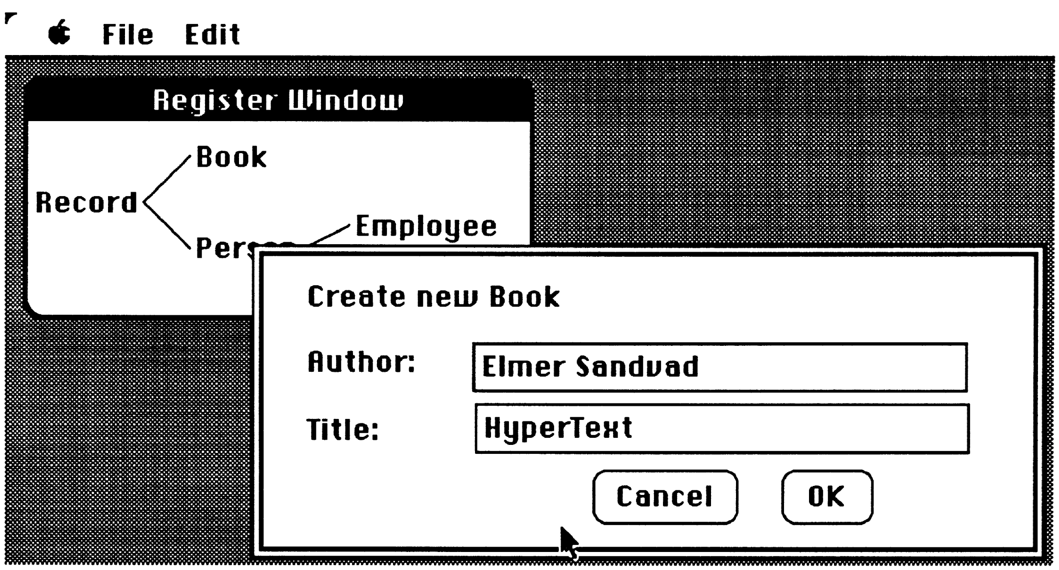


Figure 2: The dialog box createDialog

The above program fragments do not specify the Record application totally. In order to complete the program, the behavior of the other nodes needs to be specified. However, these share almost the same structure as the Book node and they are therefore excluded here. It should also be noted that checking for various (obvious) error conditions have also been ignored to reduce the code size. For the same reason, we have not put any emphasis of the graphical layout (visual appearance) of the graph in the top level window. However, including these aspects in the final program is straight forward.

4 Conclusion

As illustrated above, rather complex user interfaces with advanced application functionalities can be created with relatively little effort. What is most impost is, that the structure of the program and particularly the user intefiace structures are very intuitive and thereby supporting the creation of elegant systems.

It should also be noted that the direct mapping of object functioned onto the user interface brings the object-oriented design philosophy into both ordinary application programming and user interface programming, resulting in very uniform systems with respects to the overall structural properties of the programs.

It is important to stress that teaching object-oriented programming is not only a matter of teaching the students how to write programs using a particular object-oriented language or system. The courses must take the fundamental aspects of object-orientation seriously, and discuss concepts, their structure and relations, and the relations between “the real world” and the object-oriented models hereof in order to emphasize that yet another language does not solve any problems alone — only new approaches to problem solving and model construction can lead to better overall system capabilities.

The BETA Macintosh System is an effective vehicle in teaching object-oriented programming, since the BETA language contains language consructs directly designed to enable the construction of effective solutions to software construction using “state-of-art” object-oriented language constructs. The language consists of very few concepts and the power of the language is the orthogonality of these concepts. The BETA language is part of the Mjølner BETA System, offering several supporting tools, and the BETA Macintosh System offers the full interface to the entire Macintosh Toolbox, putting the construction of highly interactive object-oriented applications within the reach of student assignments.

The BETA Macintosh System is available as a prerelease from Mjølner Informatics at the time of writing. During the spring and summer, Mjølner Informatics will finalize the system and deliver a special educational package, consisting of the system with manuals, course outline, and supplementary material (such as the book on the BETA language).

Acknowledgements

Many people have taken active part in the development of the Mjølner BETA System and the BETA Macintosh System and we would like to thank all of them. However, special thanks are due to Apple Computer Inc. who have supported the work on the BETA Macintosh System.

References

[1] H. Abelson, G.J. Sussman, J. Abelson: *The Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] A. Goldberg, D. Robson: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[3] O-J. Dahl, B. Myrhaug, K. Nygaard: *(Simula 67) Common Base Language*, Publication N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, Oct. 1970 (Revised version, Feb. 1984.)

[4] J. Lindskov Knudsen, K. Stougaard Thomsen: *A Conceptual Framework for Programming Languages.* Technical Report DAIMI PB-192, Computer Science Department, Aarhus University, April 1985.

[5] J. Lindskov Knudsen, O. Lehrmann Madsen, C. Nørgaard, L. Bak Petersen, E. Sørensen Sandvad: *An Overview of The Mjølner BETA System*, MIA report, Mjølner Informatics, March 1990.

[6] B. Bruun Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language — Part 1: Abstraction Mechanisms — Part 2: Multi-Sequential Execution. In: B.D. Shriver, P. Wegner (eds.): *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[7] B. Bruun Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language.* Draft book, March 1990.

[8] O. Lehrmann Madsen, J. Lindskov Knudsen: Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages. In: *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP'88), Oslo, Norway, August 1988.

[9] O. Lehrmann Madsen, B. Møller-Pedersen: Virtual Classes — A Powerful Dimension in Object-Oriented Programming. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA'89), New Orleans, Louisiana, October 1989.

[10] B. Meyer: *Object-oriented Software Construction*, Prentice Hall, 1988.

[11] B. Stroustrup: *The C++ Programming Language*, Addison-Wesley, 1986.