ISSN 0105-8517

Reusability and Tailorability in the Mjølner BETA System

Claus Nørgaard
Elmer Sandvad

DAIMI PB – 300
January 1990

Reusability and Tailorability in the Mjølner BETA System *

Claus Nørgaard
Mjølner Informatics ApS, Science Park,
Gustav Wieds vej 10, DK-8000 Aarhus C, Denmark
Phone: +45 86 20 20 00 — Email: cn@mjolner.dk

Elmer Sandvad
Computer Science Department, Aarhus University,
Ny Munkegade 116, DK-8000 Aarhus C, Denmark
Phone: +45 86 12 71 88 — Email: essandvad@daimi.dk

Abstract

Reusability and tailorability in software development are discussed in general and a set of techniques supporting these concepts are described. An important concept that can give a high degree of reusability as well as tailorability is generality. Object-oriented languages give good support for expressing generality. The techniques discussed in this paper have been developed during the construction of the Mjølner BETA System. Some of them are of general (language independent) character but a great part of the flexibility in the system is due to the BETA programming language. An exception handling technique is presented as a special kind of tailoring. A general communication model supporting integration and tailorability of software components is presented.

Keywords:
Reusability, Tailorability, Generality, Object-Oriented, Programming Environments

*Presented at TOOLS '89, Technology of Object-Oriented Languages and Systems, Paris, November 13-15, 1989

1 Introduction

In software development there is currently a lot of interest in reusability. One of the reasons for this interest is the wish to reduce the total software life-cycle cost. When using traditional programming languages like Pascal and C, programmers often have to reinvent the wheel many times. In many applications there is a need for general components like linked lists, hash tables, sort functions etc., but in the actual use there is typically need for small modifications to adapt the components to a specific usage. Because traditional programming languages do not allow the programmer to express such modifications, it is often necessary to start from scratch when a general component is going to be reused.

Object-oriented programming languages give better support for reuse of software. Reusability is often seen used as a heavy argument for choosing an object-oriented language.

In this paper reusability and the related concept: tailorability are discussed. In a given context a software component is considered *reusable* if it can be used without duplicating or modifying its source code. One important concept that provides a high degree of reuse is *generality.* Object-oriented programming languages give good support for generality.

A software system is *tailorable* if the user is allowed to modify the system in order to adapt the system to specific needs. Like in reusability, generality can increase the degree of tailorability. In tailorability the emphasis is on specializing a general component to specific needs.

Reusability and tailorability are two different but strongly connected concepts. Tailoring is often connected to reuse because a component that has been used in one context seldomly fits into another context without modifications. On the other hand it is hard to imagine tailorability without reuse, because the very idea of tailoring is to adapt an existing system to specific needs. Both concepts are however justified, because the main emphasis can be on either reuse or tailoring. Examples of situations where the main emphasis is on reuse are using a queue, list, stack or hash table. Examples of situations where the main emphasis is on tailoring are adding facilities to a text editor, like syntax check or support for hypertext.

In this paper our experiences with reusability and tailorability are presented. The experiences are from the development of an object-oriented programming environment, the Mjølner BETA System. A set of techniques that support reuse and tailoring are presented. Some of them are general and can be supported by most object-oriented programming languages. One example is a communication model that supports integration and tailorability of software components. Other techniques would not have been possible if another language than BETA had been used. BETA gives good support for expressing generality. Actions (procedures, methods) and substance (type, class) can be specialized in a uniform and flexible way.

The Mjølner BETA System [1] is an object-oriented programming environment, primarily aimed at supporting development of large production programs written in BETA [2]. The environment contains a set of grammar based tools including a metaprogramming system [9], an integrated text and structure editor and a fragment system.

BETA is an object-oriented programming language in the Simula 67 tradition [13] with respect to block structure, static name binding and compile-time type checking. This is in contrast to Smalltalk [14], that has a flat set of definitions (classes), dynamic name binding and run-time type checking. The techniques described in this paper are based on 3 years experience with programming in BETA.

The structure of the paper is as follows: first the concepts reusability and tailorability are discussed in general, then the basic techniques supporting these concepts are described and illustrated by examples from the Mjølner BETA System. The techniques are specialization of actions, values and substance. Then exception handling is discussed and finally a communication model is presented, that supports integration as well as tailorability in the Mjølner BETA System.

[1]The Mjølner BETA system [12] is one of several results of the Mjølner project [11] that was a joint Nordic venture developing object-oriented programming environments. The Mjølner project was partly funded by a grant from the Nordic Fund for Technology and Industrial Development. In this paper only the Danish subproject: the Mjølner BETA system will be described.

Part of the Danish subproject has also been supported by a grant from the Danish Natural Science Research Council, FTU Grant No. 5.17.5.1.25.

2 Reusability

When discussing reusability in this paper the emphasis will be on reuse in software construction, i.e. reuse of program parts as opposed to reuse of ready-made programs like UNIX tools.

In a given context a software component is *reusable* if it can be used without duplicating or modifying its source code.

The motivation for this definition is:

- If the source code is copied, the programmer of course does not have to reinvent existing components, but a maintenance problem arises when errors are fixed or new functionality is added (in the original as well as in the copy). In addition the size of the system is not decreased.
- Like duplication, modification of a reused component would also create maintenance problems, especially for the other uses of the original component. This does not imply that a reusable software component is restricted to be one-purpose. But if the component is adapted to a specific application it must be done without modifying the original component.

Adapting an existing component to a specific application is covered by the concept tailorability.

Reusability is a relative concept. A large group of software components is reusable according to the definition above. But the degree of reuse can be very different. The more contexts a software component can be used in, the higher the degree of reuse. One important concept that can provide a high degree of reuse is *generality*. The question is: how does the programming language support generality. A software component is general if the establishment of some of its qualities is deferred. Such qualities are called *deferred qualities*. Deferred qualities must be established before the component is used. Different specializations of the general component may establish deferred qualities differently.

An example of a software component that can be more or less reusable is the list data structure. A low degree of reusability is provided in an implementation that has a fixed maximal size and a fixed element type.

A high degree of reusability is provided in an implementation that has no restrictions on the size of a list and no advance requirements on the element type. In a general list the deferred qualities could be the maximal number of elements (if not a dynamic list), the element type and some of the operations on the list.

Inheritance is widely recognized as supporting generality and most object-oriented programming languages support inheritance in one way or another. The difference between the individual languages with respect to inheritance lies in what type of qualities that can be deferred and how the qualities are established.

In Smalltalk inheritance is supported by classification hierarchies and deferred qualities are expressed by methods that are bound dynamicly at run-time. In BETA inheritance is supported by classification hierarchies too. In addition inheritance for procedures is allowed. Deferred qualities can be expressed by virtual procedures as well as virtual classes. The virtual class concept is discussed in [10]. Establishment of deferred qualities in BETA (binding of virtual procedures or classes) is done at compile-time.

3 Tailorability

A software system is *tailorable* if the user is allowed to modify the system in order to adapt the system to specific needs. Tailoring can be done by specializing the behavior of the system or by adding functionality to it.

This definition is inspired by [16]. They define tailorability to be a sub-concept of adaptability. A system can be adaptable in four ways: it can be flexible, parameterized, integratable or tailorable. In this paper we focus on tailorability. An example of parameterization is setting switches that choose between a range of different alternative behaviors. This primitive kind of adaptability is not discussed in this paper.

If the user is a non-programmer tailorability is restricted to specializing behavior and even in this case the user has limited possibilities, like combining commands in a macro-like language (e.g. spread sheet systems or database systems). When discussing tailorability in this paper the emphasis will be on tailoring by programming.

Like reusability, tailorability is a relative concept. A high degree of tailorability is provided if the amount of effort required to specialize behavior or add functionality is relatively small. As in reusability, generality can increase the degree of tailorability. In reusability the main emphasis is on setting up the general structure and isolating and deferring the qualities that might vary in the different specializations. In tailorability the main emphasis is on specializing behavior by establishing the deferred qualities or on adding functionality (qualities). Specializing behavior can be classified in: 1) specializing value (constants) 2) specializing actions (procedures, methods) and 3) specializing substance (types, classes). These different kinds of specialization are discussed in section 4.

One way of supporting tailorability is to provide the whole system to the user including the source code. But this naive solution is impractical of several reasons: 1) security concerns: the "author" of the system does not want the user to destroy anything 2) copyright concerns 3) maintenance concerns and 4) the user is only interested in functionality, the whole system can be hard to understand and even harder (impossible) to modify.

Another way of supporting tailorability is to provide a well-defined programming interface to the system. The part that the user has added to an existing system is called its *extension*. A programming interface can give a clear separation of the existing component and its extension. The user can use the system without modifying it. A programming interface can thus support reusability too. In order to fully support tailorability the interface must be a two-way interface. It might be necessary to allow the users of the component to be notified when certain events occur. This information must be transferred to the extension in some way, but at the time the component is created the users of it may not be known. One example is extending a structure editor with an incremental static semantic checker. When a part of the program is modified the checker must be told about the changes. It is thus impossible to reuse the syntax directed editor, if it is not capable of reporting these changes.

In the Smalltalk system this kind of communication is supported by sending an update message from an object to its dependents. In Lisp systems [7] [4] so-called hooks are used to let the user influence the program execution. A hook is simply a user-modifiable Lisp function that is called every time a certain point in the execution is encountered. In Mjølner BETA System the virtual procedure is used for communication from an object to its extensions. The communication model presented in section 6 uses the virtual concept heavily.

4 The basic Techniques

This section illustrates the basic techniques of BETA that supports reuse and tailoring. For a detailed description of BETA see [2] or [3].

Before we start, it is neccessary to give a brief introduction to the language. A BETA program execution is a collection of objects. An object is statically or dynamically created as an instance of a so-called *pattern*. The pattern is an abstraction mechanism that unifies type, function, procedure and class.

Due to the pattern concept, the size of the language is relatively small. At the same time however it implies that type, function, procedure and class definitions have the same syntax, which might confuse unexperienced users of BETA.

In the hope that more readers will understand this paper, we will use the modified syntax of BETA also used in [10]. Using this syntax there is syntactical difference between declaration of patterns used as procedures, and patterns used as classes. "Real" BETA can be obtained by a syntactical replacement (removal) of keywords.
An example of a pattern used as a class is

```
CC: class C (#  Decl1; Decl2; ...; Decln #)
```

This declares a class CC with the super-class C, and with the declaration of local attributes Decl1 ... Decln.
The form of a pattern used as a procedure is

```
PP: proc P
  (# Decl1; Decl2; ...; Decln
  enter In
  do Imperatives
  exit Out
  #)
```

Here the name of the procedure is PP with super-procedure P and the local attributes Decl1 ... Decln. The enter-part `In` describes the input-parameters, `Imperatives` describes the actions to be performed, and `Out` describes the output parameters.

Attributes can be (local, statically created) objects, object references (to local or non-local objects) or patterns. Object references corresponds to instance variables in Smalltalk. Allowing local patterns implies that classes and procedures can contain local classes and procedures.

4.1 Specializing Actions

Generality of a software component was defined as deferring the establishment of some of its qualities. This section illustrates how virtual procedures can be used to defer and establish a certain kind of qualities, namely actions.

Fig. 1 shows a classification hierarchy that is taken from the the window package [2] of the Mjølner BETA System.

`TitleTool` *is* a `window` that is used as a title bar. It *has* a local object (`title`) which is a static instance of class `text`. A `windowTool` *is* a `window` that *has* a local object (`titleBar`) which is an instance of class `titleTool`. `ScrollTool` *is* a `window` that is used as a scroll bar. A `scrollWindowTool` *is* a `windowTool` that *has* two local objects (`horizontalBar` and `verticalBar`), which are instances of class `scrollTool`.

The keywords @| denote that `titleBar`, `horizontalBar` and `verticalBar` are statically created as co-routines, i.e. they have their own execution thread.

In the description above the use of the verbs *is* and *has* (in italic) were used very carefully. The verb *is* really means 'is a specialization (subclass) of' or 'specializes' and the verb *has* really means 'has as local object' or 'aggregates'. The classification and aggregation hierarchy is illustrated graphically in Fig. 4. Subclassing is illustrated by means of vertical concatenation of rectangles. Aggregation is illustrated by means of nesting of rectangles.

[2]This is one of the cooperative results of the Mjølner project. The specification of window package has been carried out by the Norwegian subproject [5]. An implementation exists in Simula and BETA.

```
window: class (# ... #);
titleTool: class window
  (# title: @ text;
     refresh: extended proc
       (# do "draw this title"; INNER #);
     ...
  #);
windowTool: class window
  (# titleBar: @| titleTool;
     refresh: extended proc
       (# do ... titleBar.refresh; INNER #);
     ...
  #);
scrollTool: class window
  (# refresh: extended proc
       (# do "draw this scrollbar"; INNER #);
        ...
  #);
scrollWindowTool: class windowTool
  (# horizontalBar,
     verticalBar: @| scrollTool;
     refresh: extended proc
       (#
       do ...
          horizontalBar.refresh;
          verticalBar.refresh;
          INNER
       #);
     ...
  #)
```

Figure 1: Part of the Window Package

```
window: class
  (# position: @ point;
     move: proc
       (# delta: @ point
       enter delta
       do (delta.x + position.x,
           delta.y + position.y)
           -> (position.x, position.y);
          refresh;
       #);
     refresh: virtual proc
       (# do "draw this frame"; INNER #);
     keyPressed: virtual proc
       (#  ch: @ char enter ch do INNER #);
     buttonPressed: virtual proc
       (# do INNER #);
     ...
  #);
```

Figure 2: The Window Class

Some operations that can be performed on windows are general and can be described in the same way for all window objects. An example of this could be `move`, that simply updates a `position` attribute. `Move` is defined as a normal procedure (See Fig. 2). Other operations are specific to the actual specialization of a window. An example of this is `refresh`. If a window is uncovered it must be refreshed. But how a window is redrawn depends on the actual specialization (titlebar, scrollbar, textwindow ...). Describing `refresh` as a virtual procedure defers the description, of how to refresh, to the actual specialization of `window`. Other examples are the interactions with a window like pressing a key or clicking with the mouse in a window.

The keyword `virtual` denotes a definition of a virtual pattern. In the example above the procedure-pattern `refresh` is defined as a virtual procedure, meaning that the specification of some of the actions are deferred until later. The `INNER` keyword is known from Simula, it means that control is given to bindings of the virtual procedure (if any) in specializations of `window`. The virtual procedure was invented in Simula. In Simula a binding of a virtual procedure overwrites an earlier binding. In BETA a virtual procedure can be further bound in all specializations (subclasses) of `window`. This capability makes it possible to express the propagation of events to specializations in a nice way. The BETA implementation of the window package utilizes this possibility heavily.

An important characteristic is the way control is flowing between the `refresh` procedures at the different levels of specializations. In BETA the control is flowing from the definition of the virtual procedure downwards to the actual specialization level. The `INNER` construct directs the control flow to the next subclass that has a further binding. In Smalltalk the control begins in the actual specialization. If control is wanted to go upwards to superclasses, this must be specified explicitly at each level. The keyword `extended` denotes a further binding of a virtual pattern.

```
(# myWindow: @| scrollWindowTool
do ...
   myWindow.refresh;
   ...
#)
```

Figure 3: Activating the Refresh Chain

Consider the simplified example of how the window system might activate the refresh event in Fig. 3. `MyWindow` is a statically created co-routine. The `refresh` procedure of `myWindow` is called and the following scenario takes places during execution (See Fig. 5. The graphical notation of Fig. 4 has been extended by showing one of the procedures in all classes, namely the `refresh` procedure. A coarse hatching indicates a virtual procedure definition and fine hatching indicates a virtual binding. No hatching indicates a normal procedure.):

1. according to the class `scrollWindowTool`, `refresh` is a binding of a virtual procedure defined in the superclass chain: `scrollWindowTool -> windowTool -> window`. `Refresh` is defined in `window`. All this is determined at compile-time

2. in `refresh` of class `window` the window frame is drawn and because of `INNER`, control is transferred to `refresh` of `windowTool`. (Arrow 1)

3. in `refresh` of `windowTool` the `refresh` procedure of `titlebar` is called. (Arrow 2) This procedure is executed following the principles described in this scenario. On return, `INNER` transfers control to `refresh` of `scrollWindowTool`. (Arrow 4)

4. `scrollWindowTool` calls the `refresh` procedures of its scrollbars. (Arrows 5 and 7) On return `INNER` causes no transfer because we are "back" in the class where we started.

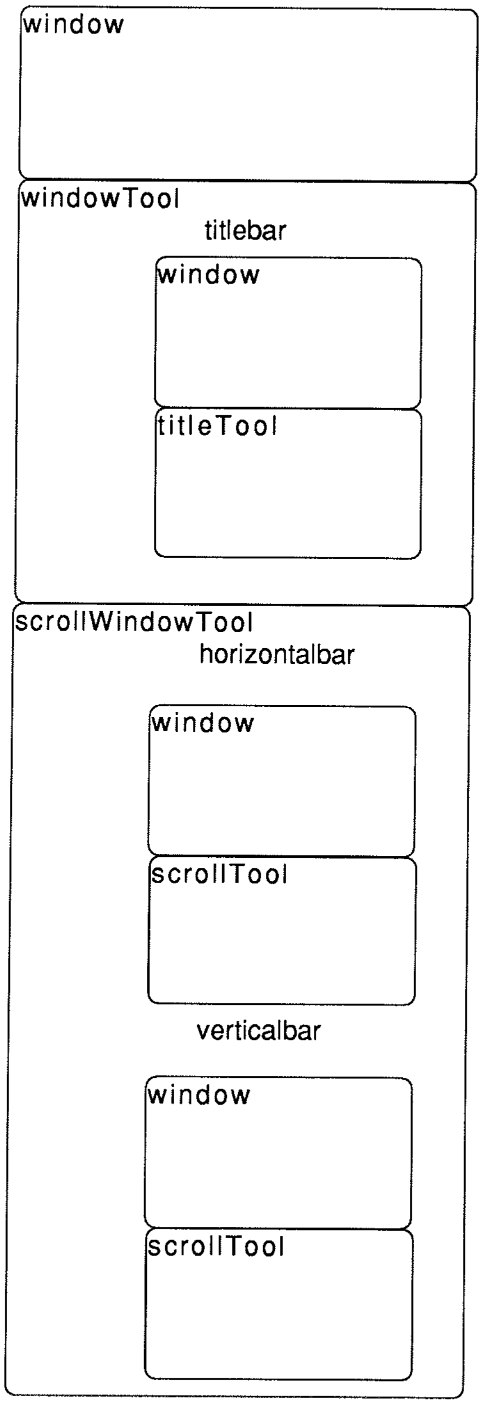

Figure 4: The Classification and Aggregation Hierarchy of Windows

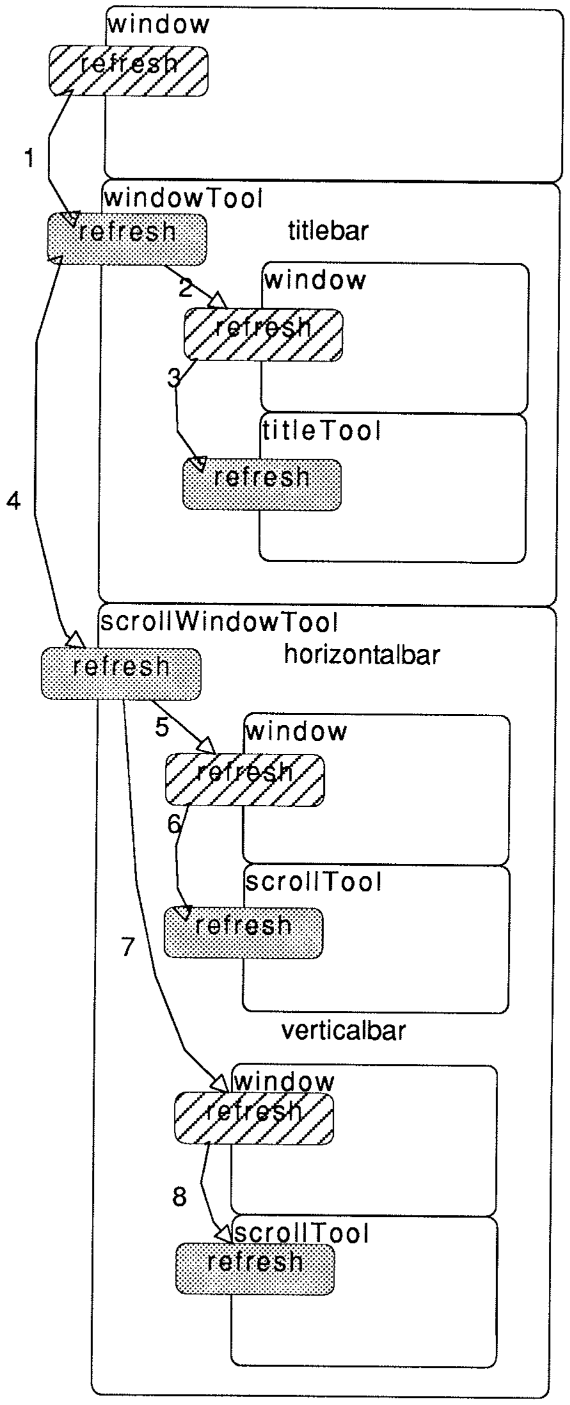

Figure 5: Control Flow when calling 'refresh'

Summarizing this section, the window package is an example of a high degree of reuse. Tools (in this example windows) are constructed in a classification hierarchy with local objects and deferred qualities, setting up a generality frame that provides reusability. At each level of specialization some deferred qualities are established without hindering further specification, and thereby supporting tailorability at each level.

The example illustrates the implementation of the window package but at the same time it is the framework (programming interface) that is offered to the user. Not only the implementors of the environment but also users can reuse the tools in the classification hierarchy in the same way as described in this section. An example is creating an instance of some specialization of a window. If the user is interested in knowing when a key on the keyboard is pressed and what the key is, she just has to make a further binding of the `keyPressed` procedure.

Using virtual procedures in the manner described in this section is not unique to the window package. The technique is used again and again in the Mjølner BETA System in order to make components more reusable and the communication model (described in section 6) of the Mjølner BETA System is based on the ability to communicate from a class to a subclass by calling a virtual procedure.

4.2 Specializing Values

A very simple way of utilizing virtual procedures is the way values (constants) can be specialized. Consider the problem of implementing a stack in an array in a traditional language. Usually you have beforehand to determine the maximal number of elements in the stack, as you have to dimension the array. In BETA you can defer the establishing of the value of the maximal number of elements until the stack is created. This could be done like in Fig. 6, and when used, a stack could be created like in Fig. 7. `CardStack` is created as an instance of a subclass of `stack` where the value of `maxNumberOfElements` at instantiation time is computed to 52. `MaxNumberOfElements` is default computed to 10, if the procedure is not bound. (In "real BETA" this example does not look that clumsy, because there are fewer keywords).

```
stack: class
  (# element: ...;
     maxNumberOfElements: virtual proc
       (* returns the value of max, default is 10 *)
       (# max: @ integer
       do 10 -> max;
          INNER
       exit max
       #);
     stackImplementation:
       [maxNumberOfElements] ^element;
       (* declaration of an array with maxNumberOfElements,
          where the element type is a reference to an element
       *)
     top: ...
     push: ...
     ...
  #)
```

Figure 6: A General Stack

```
cardStack: @ class stack
  (# maxNumberOfElements: extended proc
       (# do 52 -> max #)
  #)
```

Figure 7: A Stack of Cards

4.3 Specializing Substance

Virtual procedures were used to make specializations of actions and values. Virtual classes can be used to make specializations of another aspect of behavior, namely substance. As far as we know, the virtual class concept has not been seen in any other language. Virtual classes give good support for reuse, because it is possible to express very general data structures. Virtual classes may be seen as an alternative to generic types found in Ada [1] and Eiffel [17]. Fig. 8 illustrates how a general list can be expressed in BETA. List is a class with some attributes: the type of the elements, a pointer to the first and last element in the list and two operations: `insert` and `scan`.

Element is defined as a virtual class. It is defined to be "at least" an `object`, which means that an actual element must be an instance of class `object` or a subclass of it. `Object` is a superpattern of all patterns in BETA. Defining the element type as a virtual `object` means that the list can be bound to contain objects of any type.

`ListElement` is a private pattern used to link the elements together. It is an example of using a pattern as a type. `Next`, `elm`, `first` and `last` are object references.

`Insert` and `scan` are patterns used as procedures. `Insert` has an input parameter which is of type `element`. The `insert` operation creates an instance of the `listElement`, assigns the input parameter to the `elm` attribute, which is a reference to an `element`, and connects the listElement to the list by means of the `next` attribute.

The `scan` operation is shown in full detail because it illustrates how iterators can be specified in BETA. It also demonstrates inheritance for procedures. In the actual use of the `list` class it will be shown how this `scan` operation can be tailored to a specific purpose. `Scan` traverses the list from the start element `first`. `Loop` is a label that is used to express control flow. When an element has been processed the if-imperative is repeated, by means of `restart loop`. If the expression evaluates to false the loop terminates. The `INNER` construct is very important in the `scan` operation. Every time a new element is encountered `INNER` is called. This means that control is directed to a possible subpattern of `scan`. This is a very powerful way of combining actions.

```
list: class
  (# element: virtual class object;
     listElement: class
       (# next: ^ listElement;
          elm: ^ element
       #);
     first,last: ^ listElement;

     insert: proc
       (# e: ^ element
       enter e[]
       do ...
       #);

     scan: proc
       (# thisElm: ^ element;
          listElm: ^ listElement;
       do first[]->listElm[]
          loop:
          (if (listElm[] <> none)
           //true then
             listElm.elm[] -> thisElm[];
             INNER;
             listElm.next[]->listElm[];
             restart loop
          if)
       #)
  #)
```

Figure 8: A General List

Fig. 9 illustrates how the list can be tailored to a specific use. The `editorList` is used in the system to contain all `editor` instances, that are currently active in the system. Each `editor` instance has a window, menus and a program fragment as the most characteristic attributes.

The `editorList` is created as a static instance of a subclass of `list`. The subclass is defined immediately after the `list` identifier. In this subclass of `list` the element type is specialized. The virtual class `element` is bound, to be an `editor` object. Any element inserted in the editorList must be qualified as an `editor`. This means that it must be an object of class `editor` (or a subclass of `editor`). Due to the qualification as an `editor` the `find` operation can access the attributes of `thisElm` knowing that they exist. In this case the `frag` attribute. If the `editor` class contains no such attribute the error is caught at compile-time.

In BETA, patterns can be defined in-line in the code. The imperative `search: proc scan (# ... #)` in `find` is at the same time the definition of a specialization of `scan`, and a call of it. The input parameter of `find` is a reference to a program fragment `frag` represented as an abstract syntax tree (AST), and the output parameter is a reference to an `editor` instance. If the program fragment is not used in an active editor, the reference will be `none`. `Search` is a label that is used to stop the scanning if the right editor is found. Note that `thisElm` is inherited from `scan`. Every time an element is encountered, that does not match, control is automatically directed back to the superpattern `scan`. If the right element is found, the construct `leave search` makes it possible to stop the scanning.

Summarizing this section, the `list` example illustrates an important characteristic of BETA: the elegant way to reuse and tailor substance using the same techniques as in tailoring actions. In this example the deferred quality is substance, or at least the description of substance (class). Another interesting characteristic is inheritance also for actions.

```
editor: class
  (# win: ^ | window;
     expandMenu: @ | class menu (# ... #);
     ...
     frag: ^ fragment;
     ...
     expand: proc (# ... #);
     cut: proc (# ... #);
     copy: proc (# ... #);
     paste: proc (# ... #);
  #);

editorList: @ class list
  (# element: extended class editor;
     find: proc
       (# frag: ^ fragment;
          ed: ^ editor;
       enter frag[]
       do search: proc scan
            (#
            do (if thisElm.frag[]
                   //frag[] then
                     thisElm[] -> ed[];
                     leave search
                 if)
            #)
       exit ed[]
       #)
  #)
```

Figure 9: A List of Editors

```
caughtError: proc
  (#
  do INNER;
     "dump the call stack
      and stop the program"
  #);
deleteFile: proc
  (# noSuchFile: virtual proc caughtError;
     ...
  #)
```

Figure 10: Defining an Exception

5 Exception Handling

Exception handling can be considered as a special way to do tailoring. When using a library of routines, it is unsatisfactory to let the actual routine determine what shall happen if an error occurs. Some systems just stops execution of the program and others throw an non-understandable prompt box on top the screen, requiring the user to do something. A better solution would be to let the user of a routine determine how the error should be treated. Virtual procedures are very suitable for that purpose too.

The following example is taken from the BETA UNIX Ensemble, which is an interface from BETA to UNIX facilities like files, directories, processes and pipes. See Fig. 10. The patterns `caughtError` and `deleteFile` are both used as procedures. `CaughtError` defines the default actions to be taken if an error occurs. The `INNER` construct directs control to subpatterns of `caughtError`, if any. If `caughtError` is not specialized it will just dump the call stack and stop the program execution. If `caughtError` is specialized the subpatterns can specify how the error shall be treated, and it is possible to prevent stopping the program execution, by not returning to `caughtError`. This can be done similar to the way the `scan` operation is terminated when the right element is found.

The procedure `deleteFile` has a virtual procedure: `noSuchFile`, that is called by `deleteFile` if it is asked to delete a file which does not exists. `NoSuchFile` is specified as a virtual `caughtError`. The binding of it (and thereby specialization of `caughtError`) is done by the user of `deleteFile`, as

```
do ...
   delete:
     'editor.bet' -> proc deleteFile
       (#
         noSuchFile: extended proc
           (#
           do "File does not exist"->
              errorMessage; leave delete
           #)
       #)
   ...
```

Figure 11: Handling an Exception

shown in Fig. 11. If the file 'editor.bet' does not exist, `deleteFile` calls `noSuchFile` and a prompt-box with the error message “File does not exist” is shown to the user. Afterwards the program-execution continues at the ...'s after the deleteFile call.

In summary: This example illustrates tailoring of actions by specializing local actions. The procedure `deleteFile` is generalized by deferring the action that shall be taken (`noSuchFile`) if the file does not exists. The user of `deleteFile` can establish the action and/or let the default action (`caughtError`) take place. Another example of the uniform treatment of classes and procedures in BETA, justifying the unification of these concepts.

6 The Communication Model

The way deferred qualities like actions are established in BETA by binding virtual procedures can also be considered as a communication from a class to its specializations. Tailoring of a general class is made possible because its specializations are notified when some important events occur. In the window package example the event that a window must be refreshed or that a key has been pressed might be important to the users of the window.

A communication model has been developed in the Mjølner project that is based on this basic technique [3].

The basic idea of the communication model, is for each major component (class) in the environment, to identify a set of important events that might interest other components (objects). Each important event is implemented as a virtual procedure. Whenever one of these important events occur the corresponding virtual procedure is called. Dependent components (objects) can catch this notification in a further binding of the virtual procedure. A precondition for this technique is that all dependent components are subclasses of the component. It is however not practically possible to handle all possible dependencies by such subclassing. Instead a so-called communicating subclass of each major component is added. This communicating subclass catches the notifications and distributes them to the dependent objects. In this way the knowledge of dependent objects is isolated to the communicating subclass.

The communication model is illustrated graphically in Fig. 12. Fig. 13 shows the skeleton of aggregating two software components to one, using the communication model in BETA. The usage of virtual classes in `component12` makes it possible later on to create specializations of the composed component, where the usage of more software components are added. Fig. 14 shows an example. The component `component123` is the result of adding `component3` to `component12`.

There can thus be more than one level of communicating subclasses.

In Smalltalk there is also a mechanism for expressing dependencies between objects in order to coordinate activities. In the Object instance

[3]In [15] the model is described and the use of it in the Mjølner SIMULA System [6] is illustrated.

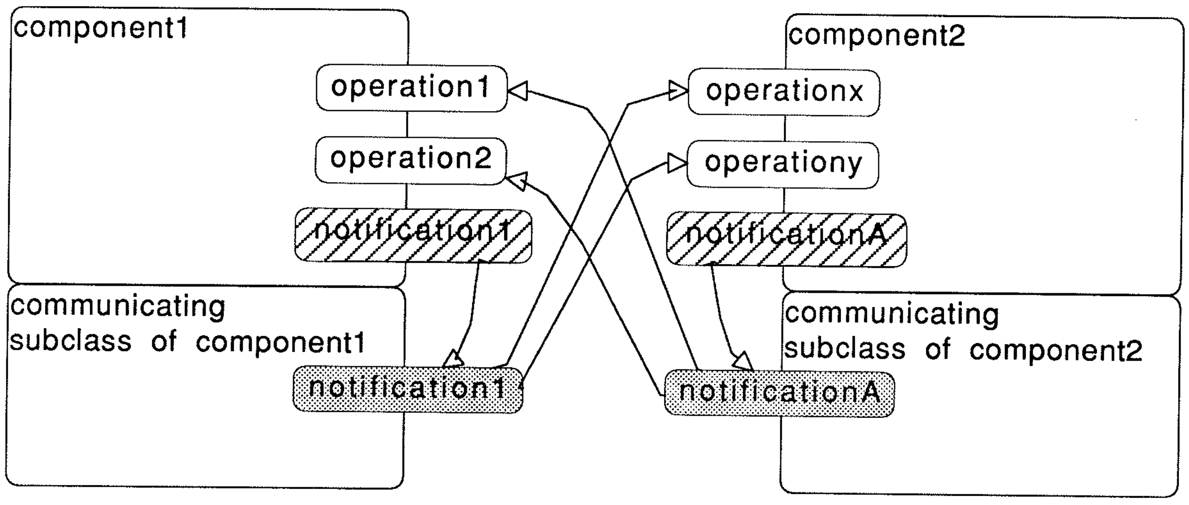

Figure 12: The Communication Model

protocol the message `changed` is provided. Each object is supposed to maintain a collection of dependent objects. If the object sends the `changed` message to itself, all dependents of the object will receive the message `update`. One parameter is allowed.

The Smalltalk approach has two drawbacks: 1) Besides the single parameter there is no distinction between the different kinds of changes that may occur to an object. 2) All dependent objects receive the update message even if they are not affected by the specific change.

In Mjølner the diversity of changes that can occur in an object is addressed by a set of virtual procedures and there is no restrictions on the number of parameters. The second problem does not occur because the communicating subclass functions as a "message exchange". It has the necessary knowledge of other objects and can distribute the notifications only to the relevant objects.

One example of using this communication model in Mjølner is the editor. Part of the internal structure of the editor has been build using this model, and the editor has been tailored to a range of different applications including an incremental static semantic checker, a transformation tool, a document structuring tool and a hypertext tool. These tools have been constructed by adding subclasses to the editor class. The notifications sent by the editor to other components in the system are also caught in the extensions to the editor.

Part of the `editorModel` looks like in Fig. 15. AST means abstract syntax tree. The notification procedure `nodeReplaced` is called by the `editorModel`

```
component1: class
  (# ...
     notification1: virtual proc
       (# ... #);
     ...
  #);
component2: class
  (# ...
     notificationA: virtual proc
       (# ... #);
     ...
  #);
...
component12 : class
  (# c1: @ commSubclassOfComponent1;
     c2: @ commSubclassOfComponent2;
     commSubclassOfComponent1: virtual class component1
       (# notification1: extended proc
            (#
            do "call operations in c2";
               INNER
            #);
          ...
       #)
     commSubclassOfComponent2: virtual class component2
       (# notificationA: extended proc
            (#
            do "call operations in c1";
               INNER
            #);
          ...
       #)
  #)
```

Figure 13: A Communication Skeleton

```
component3: class
  (# ...
     notificationX: virtual proc (# ... #);
     ...
  #);
component123: class component12
  (#
     c3: @ commSubclassOfComponent3;
     commSubclassOfComponent3: virtual class component3
       (# notificationX: extended proc
            (#
            do "call operations in c1+c2";
               INNER
            #);
          ...
       #);
     commSubclassOfComponent1: extended class component1
       (# notification1: extended proc
            (#
            do "additionally call operations in c3";
               INNER
            #);
          ...
       #);
     ...
  #)
```

Figure 14: Further Extensions

```
editorModel: class
  (#
    ...
    nodeReplaced: virtual proc
      (# newNode, oldNode: ^ AST
      enter (newNode[], oldNode[])
      do INNER
      #)
    ....
  #)
```

Figure 15: The Editor Model

```
editor : class
  (# EM: @ commEditorModel;
     PP: @ commPrettyprinter;
     UI: @ commUserInterface;
     ...
     commEditorModel:
       virtual class editorModel
       (# nodeReplaced: extended proc
            (#
            do (newNode[], oldNode[])  -> PP.update;
                newNode[] -> UI.updateMenus;
                INNER
            #)
       #);
     commPrettyprinter: virtual class prettyprinter
       (# ... #);
     commUserInterface: virtual class userInterface
       (# ... #);
     ...
  #)
```

Figure 16: The Editor Configuration

every time a node is changed in the program-fragment edited.

The editor itself is an aggregation of the three software components: the `editorModel`, the `prettyprinter` and the `userInterface` (See Fig. 16). When a subtree of the abstract syntax tree of the `editorModel` has been changed, the prettyprinter must update the screen and the user interface component must update the relevant menus.

The integration of the editor with the incremental semantic checker is done by adding a subclass to the editor (See Fig. 17). It is our experience from the development of the Mjølner BETA System, that this scheme for composing software components gives a very flexible configuration structure. It is very easy to integrate different tools and to integrate a new tool into a set of existing tools without having to change the existing tools.

```
incrementalCheckerEditor: class editor
  (# IC: @ commIncrementalChecker;
     editorModel: extended class
       (# nodeReplaced: extended proc
            (#
            do "tell the IC to reanalyze the abstract
                    syntax tree according to newNode"
       #)#);
     commIncrementalChecker: virtual class incrementalChecker
       (# ... #);
  #)
```

Figure 17: Extending With Semantic Analysis

7 Conclusion

The concepts reusability and tailorability in software development have been discussed. Generality can give a high degree of reusability and tailorability. In reusability the main emphasis is on setting up the general structure and isolating and deferring the qualities that might vary in the different specializations. In tailorability the main emphasis is on specializing behavior by establishing the deferred qualities or on adding functionality (qualities).

Object-oriented programming languages are especially good at supporting generality, and thereby supporting reusability and tailorability. BETA gives good support for expressing a general structure with deferred qualities. Virtual procedures are used to to express deferred actions and deferred values; and virtual classes are used to defer the description of substance. In this way tailoring is done in a uniform and flexible way. Beside the virtual concept another important precondition for the techniques described in this paper is that the language has block structure. Some of the examples in section 4 and 5 utilizes that it is possible to prefix a procedure with a procedure found at another block level (`scan` and `deleteFile`), and the communication model would be very clumsy without the possibility of having classes within classes.

Acknowledgement

The authors wish to thank the colleagues in the Mjølner project.

References

[1] Ada Reference Manual: Proposed Standard Document. United States Department of Defense, July 1980.

[2] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language. In: B.D. Shriver, P. Wegner (eds.): Research Directions in Object Oriented Programming, MIT Press, 1987.

[3] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object Oriented Programming in the BETA Programming Language. In: BETA Tutorial OOPSLA'89, New Orleans, 1989.

[4] R.M. Stallman: EMACS: The extensible, customizable, self-documenting display editor. In: D.R. Barstow, H.E. Shrobe, E. Sandewall (eds) "Interactive Programming Environments", McGraw-Hill,1984.

[5] T. Hauge, I. Nordgard, T. Rød, G. Ræder: Gungne Functional Specification. Project Mjølner Working Note N-EB-4.2, January 1988.

[6] G. Hedin, B. Magnusson: The Mjølner Environment: Direct Interaction with Abstractions. In: S. Gjessing, K. Nygaard (eds.) Proceedings of ECOOP '88, Oslo, Springer-Verlag, August 1988.

[7] W. Teitelman and L. Masinter: The Interlisp programming environment. Computer 14(4), April 1981. Also in: D.R. Barstow, H.E. Shrobe, E. Sandewall (eds) "Interactive Programming Environments", McGraw-Hill,1984.

[8] O.L. Madsen: Block structure in Object-oriented Programming Languages. In: B.D. Shriver, P. Wegner (eds.): Research Directions in Object Oriented Programming, MIT Press, 1987.

[9] O.L. Madsen, C. Nørgaard: An Object-Oriented Metaprogramming System. In: Proceedings of Hawaii International Conference on System Sciences - 21, January 1988.

[10] O.L. Madsen, B.M. Pedersen: Virtual Classes - a new dimension in object oriented programming. In: Proceedings of OOPSLA '89, New Orleans, October 1989.

[11] H.P. Dahle, M. Løfgren, O.L. Madsen, B. Magnusson (eds): The Mjølner Project, In: Proceedings of EUROSOFT '87, London, June 1987.

[12] J.L. Knudsen, O.L. Madsen, C. Nørgaard, L.B. Petersen, E. Sandvad: An Overview of the Mjølner BETA System. Mjølner Informatics ApS, Science Park Aarhus, November 1989.

[13] O.J. Dahl, B.Myrhaug, K. Nygaard: SIMULA 67 Common Base Language, Norwegian Computing Center, Oslo, 1984.

[14] A. Goldberg. D. Robson: Smalltalk 80: The Language and its Implementation. Addison Wesley 1983.

[15] S. Minőr: A Model for Flexible Communication Between Objects. In: Proceedings of the Simula Conference, September 1988.

[16] R.H. Trigg, T.P. Moran, F.G.Halasz: Adaptability and Tailorability in NoteCards. In: Proceedings of INTERACT '87, Stuttgart, Septemper 1987.

[17] B. Meyer: Object-Oriented Software Construction. Prentice Hall, 1988.