Paper presented at the Third Decennial Aarhus Conference, Aarhus, Denmark, August 14-18, 1995

Three Levels of End-User Tailoring: Customization, Integration, and Extension

Anders Mørch
Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
Tel: 47-22-852407; Fax: 47-22-852401
E-mail: andersm@ifi.uio.no

ABSTRACT
Tailoring is the adaptation of generic software applications such as word processors, spreadsheets, e-mail systems, and drawing editors to the specific work practices of a user organization. Tailoring shares characteristics with both design and use of computer applications, but a "design distance" separates them. An extensive literature survey and analysis of emerging trends have identified three levels of tailoring: customization, integration, and extension. They are defined, tools to support them are described, and examples are given.

KEYWORDS: tailoring, human-computer interaction, object-oriented programming, design rationale, design distance, use distance.

INTRODUCTION: IDENTIFYING THE PROBLEM

Packaged, off-the-shelf software products are dominating the software industry today because they provide standard solutions to recurring tasks performed by people in user organizations. A packaged software product is a *generic* application designed for a standard task, such as writing a paper (word processor), balancing a budget (spreadsheet), or creating a diagram (drawing editor). The tasks for which these applications are built are not well defined in advance, either because the developers have based their design on prior technology or intuition rather than on contact with real users, or because the scope of the task could not be anticipated in advance of actual use, and hence is likely to change as the application is being used. To fully utilize the potential of generic applications and prevent them from becoming rigid and static, they should be made tailorable (open and adaptable). How to accomplish this is the problem addressed in this paper. From this point of view tailoring is an activity that takes place *after* the original design and implementation of the application, but it shares characteristics with both original design and later use:

Tailoring-as-design. Tailoring is continued development of an application by making persistent modifications to the application, as opposed to the products created with it [15].

Tailoring-as-use. Tailoring is initiated in response to an application being inefficient or difficult to use for a specific task at hand.

Tailoring can begin during, or right after, installation of the application, or later during advanced use of it. It is the latter aspect that is the focus of this paper. The actors involved in the tailoring process are people in the user organization: individual users, local developers, or groups of users. Contact with original developers is assumed to be minimal, except for reporting back to the developer organization about local adaptations. This encourages developers to incorporate generally useful adaptations in future releases of the application.

A generic application is different from an application framework, but builds on it. An application framework is by definition not a usable application. It must be filled in with application-specific implementations before it can be instantiated and used, and hence is better suited as a tool for application developers than for end-users. An application framework shares the tailoring-as-design characteristic described above, but not the tailoring-as-use characteristic. Tailoring-as-use is an important condition for the success or failure of end-user tailorable applications due to its emphasis on *task-orientation* rather than *technology-orientation* A user engaged in performing a specific task becomes the "owner" of a problem when the application can no longer be used for the task. If this problem can be associated with a certain use, it will motivate an interest in understanding the problem and in learning to tailor the application to solve the problem, even if this includes learning a formal language. This situation is in many cases not a reality today. It is more common to handle breakdown situations by asking someone else or otherwise work around the problem, even if this creates suboptimal solutions. Tailoring, as presented in this paper, is an approach to helping users solve their own problems.

Tailoring is related to *adaptive maintenance* in software engineering. Adaptive maintenance is software maintenance performed to make a computer program usable in a changed external environment [7]. It is distinguished from corrective maintenance (correcting faults) and perfective maintenance (improving performance).

RELEVANT PREVIOUS WORK

This work builds on the early work in the design and implementation of tailorable systems by Trigg and his colleagues at Xerox PARC [32]. A major goal of the NoteCards hypermedia system was to make it adaptable. They defined an adaptable system to be a system where an end-user "produces new system behaviour without help from programmers or designers." They identified four ways a system can be made adaptable: (1) flexibility -- generic objects and behaviors that can be interpreted and used differently are provided, (2) parameterizable -- the user can choose between alternative behaviors, (3) integratable -- the system can be integrated with other components, internal or external, and (4) tailorable -- users are allowed to change the system itself by building accelerators, specializing behavior, or adding new functionality.

Another inspiration has been the empirical studies on the use of tailorable systems reported by Mackay [18] and Nardi [27]. Mackay's work includes data and an analysis of how users of a UNIX software environment make persistent adaptations of the system. One of her findings is that many users do not tailor (customize) their applications as much as they could. One reason for this is that it takes too much time and deviates from other activities.

Nardi's material is from users of spreadsheets and CAD packages. One of her findings is that users are able to master the formal languages embedded in these applications when they have a need and motivation for doing so. She concludes from this that tailoring-languages should be task-specific, that is, the primitives of the language should correspond to tasks of the application domain. An example is the spreadsheet formula language which has as primitives task-specific functions that map to tasks frequently performed by spreadsheet users (sum, average, etc.).

The contribution of this paper is twofold: (1) identification of three fundamental tailoring activities, and (2) addressing some of the remaining difficulties with tailoring identified by the authors above. The latter is accomplished by finding answers to the following three questions: (1) How can users tailor without direct access to original designers and programmers?, and (2) how can tailoring be better aligned with other work activities?, and (3) how can tailoring be made a human activity (independent of specific computer technologies, such as spreadsheets and hypermedia)? The short answers (my positions) to these three questions are, respectively: (1) integrating design rationale with the application, (2) tailoring after breakdown in use, and (3) task-specific indexing from presentation (task) objects to the underlying functionality implemented in a general purpose programming language. These three positions will be further elaborated in this paper. They are not meant to be a final answer on tailoring, but rather to present one perspective and to stimulate further research on the topic.

THREE LEVELS OF TAILORING

In the literature on tailoring many different concepts are used to describing tailoring activities. The most common ones are adaptation, customization, end-user modification, extension, personalization, and tailoring. They partly overlap with respect to the phenomena they refer to, and the same concepts are sometimes used when referring to different phenomena. There is also some overlap with a related area, end-user programming. End-user programming refers to the activity of writing an application program in an end-user programming language, such as HyperTalk, Visual Basic, or AppleScript. These special purpose programming languages (also called scripting languages) are interpreted languages which means that independent pieces of program code (scripts) can be written and immediately executed, thus shortening the edit-compile-debug cycle of compiled languages. Most general purpose programming languages are compiled languages.

Tailoring might include end-user programming as a technique, as in categories 2 and 3 defined below, but end-user programming is first of all a technique for creating new applications, that is, a design concept. An example of an end-user programming language is the spreadsheet language, which has formulas to create new spreadsheets (design) and macros to modify the spreadsheet application itself (tailoring).

The literature analysis reported in this paper gave rise to three categories of tailoring: customization, integration, and extension. As in any categorization, there are instances that cut across categorical boundaries. They are mentioned at the end of each subsection.

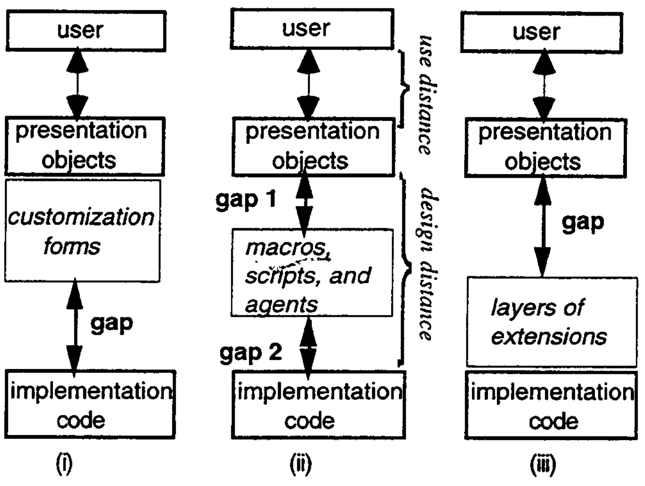
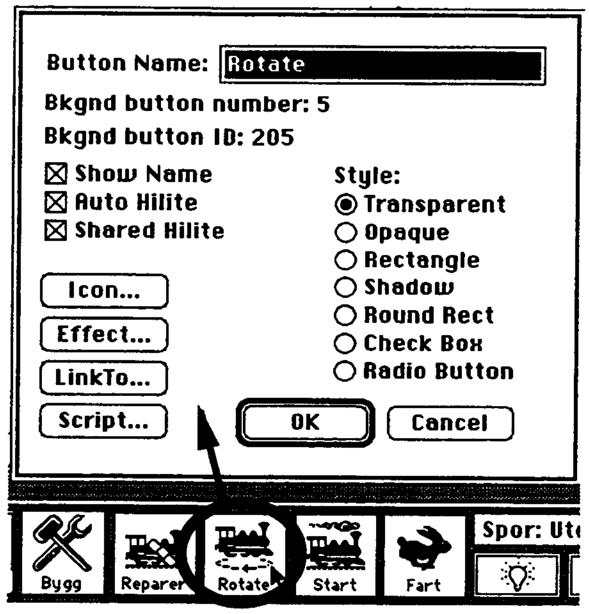


Figure 1: Tailoring by customization, integration, and extension by bridging a use distance and a design distance.

Figure 1 illustrates the three categories (levels) of tailoring. The boxes in thin lines name the means (tools and techniques) for doing tailoring. Tailoring is seen as a way to bridge the design distance between *presentation objects* accessible in the user interface and the underlying *implementation code* defining the functionality and written as text in a general purpose programming language.

Presentation objects range from simple user interface widgets such as menu items, buttons, and icons, through composite widgets such as menus, menubars, and dialog boxes, and up to high-level, application-oriented presentation objects such as tables, charts, and domain-specific symbols. They are the "handles" encountered in the user interface that mediate input and output to and from the system. Input is initiated when the user selects a presentation object with, for example, a mouse or the keyboard. Output is initiated by the system as a result of executing the functionality associated with the selected presentation object. It may create new presentation objects or change the appearance of existing ones in addition to the effects caused by executing the functionality.

The difference, or gaps, between presentation objects and implementation code can be illustrated with HyperCard (see Figure 2). From a button presentation-object there is access (via a property sheet) to a HyperTalk script that defines the functionality associated with the presentation-object (and executes it when the button is pressed and released).



Button Name: Rotate
Bkgnd button number: 5
Bkgnd button ID: 205

☒ Show Name
☒ Auto Hilite
☒ Shared Hilite

Style:
[✓ Transparent]
Opaque
Rectangle
Shadow
Round Rect
Check Box
Radio Button

Figure 2: HyperCard's property sheet provides basic tailoring options. There is access from each presentation object (e.g., button Rotate) to its functionality written as a HyperTalk script (script not shown).

HyperCard supports both customization (modifying name, style, icon, and the effect of a button) and simple integration (linking a button to a another card). The most important difference between HyperCard and the general approach to tailoring presented in this paper is that scripting languages, such as HyperTalk, are not general purpose programming languages and cannot be used to create full-fledged implementations. Languages for implementation need to support tailoring by extension in addition to customization and integration. This requires language mechanisms for specialization and virtual binding [20], and compilation in addition to interpretation [21]. As will later be shown, scripting languages are better thought of as special purpose "integration languages" than as general purpose implementation languages.

Buttons [19], which resembles HyperCard in many ways, gives the user access to implementation code written in a general purpose programming language (LISP). The buttons are presentation objects like buttons in HyperCard, but have an additional feature: they can be encapsulated in e-mail messages and other documents and passed around to other users within the Xerox LISP environment. Buttons are not meant to be inserted into application frameworks and are therefore not well suited for generic applications. Nevertheless, both HyperCard and Buttons have been important inspirations for the current work: HyperCard due to its simple design and ease of use, and Buttons because it suggests ways to integrate design rationale with basic executable program units.

Use distance and design distance

The *usability* of an application can to a large extent be determined by how well the appearance ("look & feel") of presentation objects "speak" for the use of the application. This identifies a *use distance* (see Figure 1) between the user and the presentation objects, a distance between anticipated effects (based on the goal to be accomplished) and the actual outcomes of executing commands in the interface. This distance can be divided into two (partly disjoint) mental distances: (1) semantic distance and (2) articulatory distance [16]. A goal of direct manipulation interfaces is to minimize these two distances. Two ways to shorten the semantic distance are: (1) to bring the computer closer to the user by building interfaces that aid the user in goal formulation and task execution, and (2) to bring the user closer to the computer by helping him or her to learn the interface better.

The articulatory distance is shortened by properly designed presentation objects. Examples of presentation objects with a short articulatory distance are the text style menu items **Bold**, *Italic*, Underline. The difference between the look & feel of these menu items and the effect of using them is minimized. They are therefore sometimes referred to as onomatopoeic presentation objects. An onomatopoeic word in a natural language is a word whose meaning (the referent) is imitated by the sound (a presentation object) associated with uttering the word (boom, splash, scratch, etc.). Commercial application vendors put major efforts into making icons visually onomatopoeic with respect to a target task domain in order to make them self-explanatory.

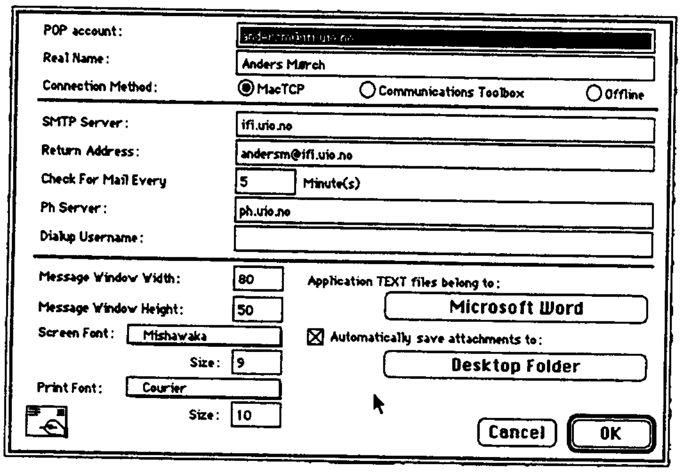
The usability of a system will improve if we can bridge the use distance because the user will experience a direct engagement in the world modeled by the presentation objects. This is important to ensure task-oriented user interfaces, but beyond the scope of this paper and therefore not addressed. The focus in this paper is to find a way to bridge the *design distance* between presentation objects and their underlying implementation codes.

The design distance prevents a user from understanding the rationale behind the implementation code underneath a presentation object [26]. Although it is different from the use distance, the design distance is also partly mental. It becomes important when the user knows how to use the application (i.e., the use distance has been bridged), but the underlying functionality is no longer adequate for the task the user wishes to perform. Therefore, knowing about the design distance and how it can be overcome is of the utmost importance for anyone wanting to understand tailorability and how it can help to improve the *usefulness* of an application.

Whereas a short use distance will improve the usability of an application, a short design distance will not necessarily improve the usefulness. There is a potential danger in narrowing the design distance by bringing presentation objects closer to the implementation code. For example, if textual presentations are automatically extracted from the source code itself (e.g., from its function names), systems with short design distance will be created -- but also with low usability. This is not the intent in this paper. On the contrary, to ensure high usability, presentation objects should align with the structure of an external task domain model (including the work to be supported) rather than with the structure of the implementation code. This may create a permanently wide design distance, but it can be bridged, and three techniques for this purpose are presented in this paper: customization forms, integration languages, and layers of extensions. When taken in their proper order (from top to bottom) they support a gradual transition into the computational complexity of an application.

Customization (Level 1)

Customization is illustrated in Figure 3. The screen snapshot shows a preference form from the Eudora e-mail system. A preference form is a dialog box in which a user can set parameters for the various configuration options the application supports. In the case of Eudora, typical options include choosing an e-mail address, fonts for reading and writing, and the frequency of updating the mailbox. The system comes with default values for many of these options.



POP account:	andersm@ifi.uio.no
Real Name:	Anders Mørch
Connection Method:	[✓ MacTCP] Communications Toolbox Offline
SMTP Server:	ifi.uio.no
Return Address:	andersm@ifi.uio.no
Check For Mail Every	5 Minute(s)
Ph Server:	ph.uio.no
Dialup Username:	
Message Window Width:	80
Message Window Height:	50
Screen Font:	Mishawaka
Size:	9
Print Font:	Courier
Size:	10
Application TEXT files belong to:	Microsoft Word
[✓ Automatically save attachments to:]	Desktop Folder

Figure 3: Customizing the Eudora e-mail system

Customization can also be used to modify the appearance of presentation objects by direct manipulation. For example, a HyperCard button-icon can be modified into a new icon or a text-string. The tools used for customization are called *customization forms* and they help to shorten the design distance by allowing users to edit attribute values of presentation objects during run-time. Customization is broadly defined as follows:

> Modifying the appearance of presentation objects, or editing their attribute values by selecting among a set of predefined configuration options.

Customization is the most common term used in the literature when describing tailoring. It is similar to what Trigg et al. [32] call parameterizable options, from which a user can choose between alternative system behaviors. Mackay [18] defines customization as the mechanisms that allow users to adapt their personal software environment without writing code, with changes that persist across sessions. The examples she gives include the setting of options such as menus, fonts, and window layout, and the creation of macros to automate repeatedly performed commands. This paper includes the latter aspect of tailoring in level 2, integration, since it is a form of end-user programming (writing or recording macros and scripts).

Cypher's definition of customization [5] is similar to Mackay's and also includes both customization and integration. He defines customization as any capability that makes a generic program suitable to a specific user need, and he gives as examples preferences (as defined above), templates (such as style templates on word processors), and automated activities (created by macros and scripts). Neither Mackay nor Cypher discuss extension, level 3.

Fischer and Girgensohn [9] identify the following four characteristics of end-user modifiability: (1) setting parameters (as defined above), (2) adding functionality to existing object classes, (3) creating new object classes by modifying existing ones (extension, level 3), and (4) defining new functionality from scratch. Characteristics 2 and 4 are not included in my definition of tailoring since they have more in common with original design and programming than with tailoring. It is argued later in this paper that an important aspect of tailoring by extension (level 3) is that end-users should be able to *add* to existing (and working) code, but not to *replace* (and possibly destroy) it, and that strongly-typed object-oriented programming languages are especially well suited for this purpose. Furthermore, Fischer and Girgensohn do not include any means for integration by scripting, such as the automation of repeatedly performed commands.

Girgensohn et al. [13] have proposed an extension to static customization forms with the concept of *dynamic* forms. This technique allows a form to be incrementally displayed by hiding irrelevant fields. Each field is an active entry similar to a spreadsheet cell and thus can have formulas attached. Furthermore, new entries (such as class attributes) can be added to the form by a high-level form description language. This allows new presentation-object classes to be created by customization.

Integration (Level 2)

Integration goes beyond customization by allowing users to add new functionality to an application. This is accomplished without accessing the underlying implementation code. Instead, users tailor an application by linking together predefined components within or across the application. Each component is a modular piece of functionality implemented to execute a well-defined task, such as a single command in a word processor, or a chart-drawing function in a spreadsheet. Integration is also a way to automate a sequence of frequently repeated commands to improve existing functionality.

Components to be integrated range from low-level commands to high-level applications, and they consist of a presentation object (command name, visual appearance) and, if applicable, an underlying implementation (the code associated with executing the component). Components are integrated by means of an advanced copy&paste operation (which copies both presentation object and functionality), or by macro and script recording. This gives rise to two kinds of integration:

1. *Hard integration*: a component is integrated by a "copy&paste" operation that results in the component becoming physically attached ("glued" or "plugged") to the application.
2. *Soft integration*: a component is integrated by means of a macro, a script, or an agent, and may execute its functionality in the context of another application.

Hard integration is based on the analogy of hardware engineering where modules (chips and boards) are plugged into sockets inside a computer [4]. The result is a seamless integration of a new component with an existing application as long as there is a well-defined protocol of communication between them. Hard integration can further be divided into two subcategories: (1) textual (compile-time) composition, and (2) binary (run-time) composition. Textual components need information about "slots" or "hooks" for integration into the implementation code (such as class names, variable declarations, and method definitions). This type of integration is closely related to "extension," and is further discussed and defined in the next section.

Binary components (such as object instances) are integrated into the run-time environment of the application. A binary component needs information about input/output parameters to be passed between itself and the run-time environment (such as in message-passing protocols). Binary components are therefore not extensible the way textual components are. They have more in common with soft integration (see below). In environments that support dynamic (run-time) interpretation of textual components, it is possible to combine the two types of hard integration [21, 22].

Soft integration is a way to *loosely* couple program executions (commands, method invocations, and object instances) and to link design rationale (a type of program documentation described below) with the application. It is most notably supported by scripting languages. Scripting goes beyond low-level macro recording in two fundamental ways: (1) by allowing high-level components to be integrated, and (2) by allowing scripts to be edited in a high-level language.

An example of a macro is an Emacs command for copying all section headings of a paper and putting them into a new buffer as an outline of the paper. Such a macro can be recorded and named in Emacs [3]. The new macro consists of a sequence of existing Emacs commands (search for heading, copy heading, create new buffer, paste heading, etc.). Macro recording makes it possible to create commands that did not previously exist in the application. However, in Emacs, it is difficult to edit a macro after it has been recorded. It is easier to abort the recording and start all over again than it is to edit the macro afterwards. This shortcoming is eliminated by high-level scripting languages that support script editing after recording. Figure 4 shows a script recorded with the scripting language AppleScript [29]. The script copies all the text of one file and places it into a new one and then saves the new file with a unique filename. This allows several versions of a paper to be saved on file.

```
tell application "Scriptable Text Editor"
 activate
 select contents of document 1
 copy
 make new document at beginning
 set position of document 1 to {97, 236}
 paste
 copy "-of-" & name of document 1 to Filename1
 copy the ticks to timestamp
 copy "copy-" & timestamp & Filename1 to
  FileName2
 save document 1 in FileName2 --a doc snapshot
end tell
```

Figure 4: A script that automates a sequence of text editing commands. The numbers (97, 236) refer to the x & y coordinates of the top left corner of the new document.

Another example of high-level component integration is a script that lets the user retrieve information from a database, use a spreadsheet to graph it, and then place the graph into a text he or she is writing with a word processor. This kind of tailoring has been described by Sumner and Stolze [30] as the "high-tech toolbelt" approach to the use of commercial, off-the-shelf applications. It may include components executing in the context of a remote host [33]. This kind of soft integration may also be possible with hard integration by "copying" the graph-drawing component of the spreadsheet and "pasting" it into the word processor, making it a seamless word processor with graph-drawing functionality [2].

We can now define soft integration as:

> Creating or recording a sequence of program executions that results in new functionality which is stored with the application as a named command or component.

Languages that create or record a sequence of program executions (most notable macros and scripts) are referred to as *integration languages*. They serve to integrate existing run-time functionality rather than to create new functionality from scratch. Scripting languages are therefore *special purpose* programming languages and not suitable as general purpose implementation languages. Integration languages help to bridge the design distance by serving as high-level (possibly application-oriented) languages that fill in part of the gap between presentation objects and implementation code.

The components linked by an integration language can also help to bridge the design distance. A special kind of component in this regard is design rationale. Design rationale is the documentation that captures the reasons behind an artifact. Most work on design rationale is based on representing *argumentation* in the form of issues, positions, and arguments [24]. The current work generalizes the concept of design rationale to include other means for representation as well: pictures, diagrams, stories, argumentation, and scenarios [26]. Argumentation is considered only one of several alternative ways to capture the reasoning behind an artifact. Design rationale, which can be linked to implementation code by means of soft integration, shares characteristics with both presentation objects and implementation code and can thus help to bridge the design distance in two different ways:

1. *From presentation and downward.* Design rationale components and presentation objects are built out of the same kind of material: text, graphics, sound, video, and animation. They are either static or dynamic.

2. *From implementation and upward.* Design rationale aligns with the structure of the implementation code at significant points in its development due to their co-evolution. Design rationale is a way to comprehend the implementation.

An alternative technique is to document a system's dynamic (run-time) behavior rather than its static implementation code. An approach to the former has been suggested by Dourish in chapter 6 of this volume [6].

Trigg et al. [32] define a system as integratable if it can be linked with other facilities within its environment or connected to more remote sources. They have used this concept to integrate NoteCards with a set of media editors (for text, graphics, bitmaps, and animation). For example, the animation card type is integrated with an animation editor that is located in the surrounding LISP environment. By allowing users to integrate new NoteCard types with corresponding types of media editors, they support tailoring by integration.

In their discussion of integration, Trigg et al. also include accelerators. They define an accelerator as "a piece of functionality that encapsulates a set of existing behaviors." This definition captures both Emacs macros and keyboard shortcuts. Both Cypher [5] and Mackay [18] include macro and script recording as part of their definition of customization (described above). At the GMD research center in Germany, a team of computer and social scientists have experimented with various ways to adapt commercial applications by means of customization and integration [28]. In particular, they have developed an extension to the spreadsheet system Excel called Flexcel. Flexcel integrates support for both adaptability (tailoring) and adaptivity (system-initiated help). Adaptivity is supported by a knowledge-based critiquing mechanism that helps users to do the adaptation. It was found to be an important resource during adaptation.

Malone et al. describe Oval [23] as a radically tailorable tool for cooperative work which allows new groupware applications to be created by combining existing objects, views, agents, and links from their tailoring language. It thus supports integration of high-level components within an application in order to create more specialized applications. Oval has no means for recording macros and scripts, but it does have a rule-based language for responding to external events, such as mechanisms for filtering incoming messages in an e-mail application.

Extension (Level 3)
Integration is well suited for tailoring tasks that do not require changes to the implementation code of an application or any of its components. It assumes, for example, that a spreadsheet drawing component has all the functionality needed for graph-drawing, and that if it doesn't, the functionality can be found somewhere else and integrated. However, this is not always the preferred solution. Sometimes components are unavailable, unsuitable, or not even working properly. They may cause fragile links among other components and thus create unstable applications that may prevent further tailoring or even the use of the application. To compensate for this potential weakness, components themselves must be made tailorable.

The kind of tailoring discussed in this section is tailoring by extension: adding new functionality to applications and textual application components. This allows *radical changes* to be made, which are changes that cannot be anticipated by developers at design time and that require changes to the implementation code itself [8, 21, 26]. Whether these changes should be done by end-users, developers, or automatically by the application itself, depends on the kinds of extensions to be made.

What implementation language to use when building extensible applications is a matter of debate. In this paper it is argued in favor of general purpose programming languages, and strongly-typed, object-oriented languages in particular. Such languages have dedicated mechanisms for creating type-safe extensions to an application without corrupting the old code. One such mechanism is subclassing (also referred to as specialization or inheritance).

Subclassing is a way to create a new class of functionality by copying and *extending* an existing class originally designed and implemented by developers and organized in a classification hierarchy (Figure 5). Subclassing is safest in strongly-typed, block-structured languages in which a subclass is not allowed to redefine the attributes and methods of its parent class, and can be placed next to its parent at the same block level. Extensions in the BETA language [20] are created by *adding* new attributes to a class, and new statements to virtual procedures (extensible methods). Extensions are added at "open points" in the application. Open points, described by Grønbæk et al. in chapter 8 of this volume [14], are identified by developers during the design of the application. They point out areas in the implementation where multiple design alternatives were discussed, but only one was actually implemented. Open points provide "hooks" for further extensions where it is possible to implement alternative designs in the future.

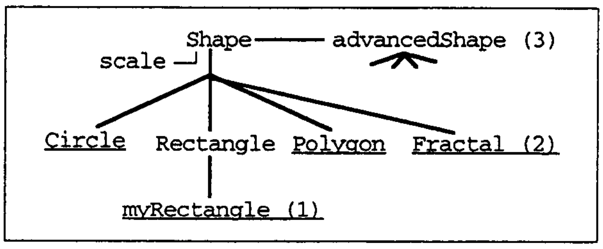


Figure 5: Classification hierarchy of a drawing program with three types of extensions: (1) simple, (2) complex, and (3) restructuring. Abstract superpatterns are in plain; concrete patterns are underlined. Scale is method (virtual procedure pattern) of class (pattern) Shape. In Beta [20], both classes and methods are patterns.

Safe extension does not mean that old code should never be discarded -- it should -- but not by end-users. Modification and replacement of existing code can be accomplished by restructuring, reseeding [11], or reorganizing [12] the system functionality. This is an activity that is done by developers, and should include the most important extensions created by tailors in the user organizations. Extension is defined as follows:

> Extension is an approach to tailoring where the functionality of an application is improved by adding new code.

When extending an application written in an object-oriented programming language and organized in one or more classification hierarchies, we identified three kinds of extensions that are illustrated in Figure 5. (The terminology used in Figure 5 is used in the remainder of this section:)

1. *Simple extension:* specialization of concrete patterns (Rectangle --> myRectangle; Rectangle --> Square).

2. *Complex extension*: specialization of abstract superpatterns into concrete patterns (Shape --> Circle; Shape --> Fractal).

3. *Restructuring*: specialization of (abstract or concrete) superpatterns into new (abstract or concrete) superpatterns (Shape --> advancedShape; Shape --> Oval).

Simple extension is extension of the patterns that can be directly accessed from presentation objects in the user interface of the application. They are normally concrete patterns (such as Circle, Rectangle, and Polygon in Figure 5). For example, to create a special type of Rectangle (such as myRectangle) we simply extend the Rectangle implementation code that can be accessed from its associated presentation object (the graphical symbol of a rectangle on the screen). If the newly added extensions preserve all the defining properties of the parent class, the parent class can be made abstract (i.e., a non-instantiated Rectangle). On the other hand, if the new extensions define a new kind of graphical shape, such as Square, we may want to keep Rectangle concrete.

When the implementation code is organized in one or more classification hierarchies some of the functionality will not be part of concrete patterns. The functionality is still available, but it is defined higher up in the hierarchy. For example, a Scale command may be defined as a virtual procedure pattern (extensible method) of a pattern Shape (Figure 5). Shape is an abstract superpattern and therefore not (directly) instantiated during a program execution. Abstract superpatterns cannot be directly accessed since they have no "handles" (presentation objects) in the user interface; there are no Shape presentation objects, only presentation objects for the concrete subpatterns Circle, myRectangle, and Polygon.

Abstract superpatterns define the general properties and overall behavior of a concept. They are specialized into operational concepts by subclassing. However, abstract superpatterns should also be accessible to the user since they provide hooks for extensions similar to the concrete patterns. There are two ways to extend an abstract superpattern: (1) through extensions that create concrete subpatterns (complex extension), and (2) through extensions that create new superpatterns (restructuring).

Support for complex extension is needed when we want to add new functionality:

- A new graphical shape (such as a fractal) that cannot be created by simple extension (Figure 5).
- A LISP function for counting the number of words in a text-editor buffer (Figure 6).
- A new kind of kitchen appliance (such as a microwave oven) to a kitchen design environment [9].
- A graphics editor to a word processor that has only text-editing capabilities.

```
(defun word-count ()
 (interactive)
 (let ((count 0))
    (goto-char (point-min))
    (while (< (point) (point-max))
       (forward-word 1)
       (setq count (plus 1 count)))
    (message "buffer contains %d words" count)))
```

Figure 6: Extending Emacs with a LISP function for counting the number of words in a text editor buffer.

What these four examples have in common is that they cannot be created in the same simple way as myRectangle. This is because the implementation code to be extended may not be accessible from any of the application's already available presentation objects. If we want to add a Fractal shape to a generic drawing program or a word-count function to a text editor, and there is nothing already in the application that can serve as a natural base, we have build more or less from scratch. Fractal, for example, can be built as an extension to the abstract superpattern Shape (Figure 5). It will be a non-trivial extension, but the new Fractal shape may not have any side effects since it becomes a leaf node in the classification hierarchy. In this case there are few or no dependent subpatterns of Shape that have to be included as subpatterns of Fractal.

If we want to extend scale, however, we have to be more cautious (Figure 5). We can, for example, specialize the pattern it is a component of (e.g., Shape --> advancedShape), which may require restructuring some of the subpatterns of Shape if we want them to inherit from advancedShape (at least giving them a new superpattern name). Some of them may also depend on the scale definition in Shape and may therefore have to be updated and recompiled to make them compatible with the scale extensions added in advancedShape.

Similarly, adding a new kind of shape, such as Oval (subpattern of Shape, superpattern to Circle) may also require restructuring. To be generally available, such a pattern should be added as a superpattern and thus may require restructuring and recompilation of dependent subpatterns (such as Circle). Automatic restructuring by creating new superpatterns based on factoring out common (e.g., duplicated) attributes and methods of same block-level classes, without adding any new functionality, is not considered an extension. Automatic restructuring may, however, improve the application in other ways, such as optimizing its performance.

Introducing new classes of functionality (e.g., Square, Fractal and Oval) will usually require tailoring the user interface as well (e.g., new menus, menu items or palette entries). This can be accomplished by customization, as described in a previous section, or by simple extension, since presentation objects are defined as concrete patterns.

Tailoring by complex extension and restructuring is much more demanding then tailoring by simple extension. However, in some cases complex extension can be automated by the computer. If, for example, the functionality to be added has already been built (e.g., as a component of another application) and has a well-defined protocol for communication with other applications (i.e., "hooks" or "slots" are known and part of the calling application), it can be "plugged" into the calling application by a "copy&paste" component operation [12, 17].

Complex extension was identified as "hard integration" in the previous section. Whether this form of tailoring should be thought of as integration or as extension depends on how smoothly the "plug-in" operation can be performed. A short answer to this is that if a "plug-in" can be performed by an end-user it should be thought of as integration, and otherwise it should be thought of as extension. Advanced forms of extensions (complex extension and restructuring) require design expertise and should be left to professional developers or experienced end-user programmers.

The third and final way to bridge the design distance between presentation objects and implementation code is to add layers of increasingly domain-oriented extensions on top of an implementation base layer. This is illustrated in Figure 7, which shows the extension layers in Janus, a domain-oriented design environment for kitchen floorplan design [10]. It allows a user to manipulate functionality at various levels of abstraction [9].

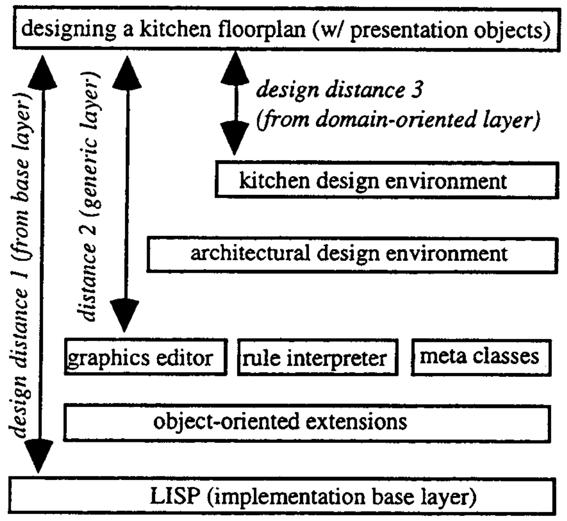

designing a kitchen floorplan (w/ presentation objects)
design distance 1 (from base layer)
distance 2 (generic layer)
design distance 3
(from domain-oriented layer)
kitchen design environment
architectural design environment
graphics editor
rule interpreter
meta classes
object-oriented extensions
LISP (implementation base layer)

Figure 7: Bridging the design distance by extending an implementation base layer with layers of increasingly domain-oriented abstractions. Design distances are measured against the respective slayers. Adapted from [9].

Extension layers help to shorten the design distance by either "lifting" the implementation base by increasingly domain-oriented layers, or by "dropping" the presentation objects down toward the implementation code. The latter approach was discussed in the beginning of this paper: it is only satisfactory when the application is meant to have interfaces to lower level substrates (graphics editing, implementation code editing, etc.). It is not acceptable as an interface to end-users in general. For example, creating a kitchen floorplan with a generic drawing editor is tedious; creating the floor plan in LISP is unacceptable.

Other systems that incorporate techniques for extension associate a specialized language with each extension layer. Examples are SchemePaint [8], Oval [23], and ACE [27].

Supporting tailoring by extension in a statically-compiled language, such as BETA [20], may seem like overkill since it requires a recompilation of a large portion of the code for each extension to be added. Simple extensions will usually include no more than small pieces of code which can be quickly interpreted and directly inserted into the previously compiled code. A solution to this problem has been proposed by Malhotra [21, 22] and is an important foundation for this work.

GENERAL DISCUSSION

Is it likely that end-users will tailor their applications to the extent envisioned in this paper? Most commercial applications already allow a significant amount of customization; word processors, for example, have configuration options for fonts and formatting styles. Some of this is far from trivial, but it can be simplified if customization issues are given high priority from the beginning of a system development project. Integration may be the next new wave of application/component building, end-user programming, and tailoring. Many of today's applications are too complex to be maintained by single vendors. Distributing specialized functionality among component builders will make it possible to tailor generic applications by integrating individually developed specialized application components.

Tools that support tailoring by extension are not a common feature of commercial applications today, although tailoring by extension is, more or less, a standard way of doing things in object-oriented programming environments. Making this strategy available to end-users of generic applications will require additional support mechanisms, some of which has been suggested in this paper.

Another argument in support of tailoring by extension is that this approach to (evolutionary) programming may be a good way to teach object-oriented programming. Extending a usable application gives students a relevant working example. If the teacher provides a properly designed generic application, all language constructs of the programming language can be taught by example, and students will be able to create their own unique solutions as personal extensions to something that already works. This can be accomplished with no more overhead than understanding the application domain (which, of course, will vary in complexity from domain to domain), and the students solutions can be compared to each other.

FURTHER WORK

The tailoring techniques presented in this paper can be extended by adding support for organizational changes, such as work redesign and learning [28, 31]. This is consistent with the Scandinavian approach to system development, which places an equal emphasis on technical and organizational issues [1]. This paper has focused almost exclusively on the technical issues, but it is not incompatible with the broader approach since the individual end-users are given first-class status. A similar effort should be undertaken from the perspective of organizational redesign.

An important distinction between tailoring and traditional systems development is that tailoring does not follow a linear analysis --> design --> implementation model. Bødker and Trigg [31] have suggested reversing the arrows in the traditional model, allowing design to inform implementation during tailoring. For this to be possible, design representations must be accessible to users during use, and therefore must have been integrated into the system during development. This requires not only a new model of tailoring but also an extended model of systems development. The author has suggested an approach to the former with application-units. They have design rationale as an integrated part [25].

Some design questions for further investigation are:

- Can scripting languages be used in the implementation to create more domain-oriented layers of extensions?
- Should integration languages as a rule be interpreted rather than compiled?
- Which is the lesser evil "fragile links" (a problem of integration) or "fragile superclasses" (a problem of extension)? How can they be overcome?
- What are the relevant issues to be addressed when choosing between a statically-typed language such as BETA and a dynamically-typed language such as LISP to serve as a base for implementing extensible applications?

ACKNOWLEDGMENTS

Members of the Systems Development Group at the University of Oslo provided important feedback during the research reported in this paper. Joan Greenbaum and Frieder Nake provided comments on the first draft. Odd Steen and Trevor Wood-Harper provided comments on the conference proceedings version. This paper is part of the project "Methods and Tools for Tailoring of Object-oriented Applications", which is supervised by Kristen Nygaard, and supported financially through a Dr.scient stipend by the Research Council of Norway under grant nr. 100618/410.

REFERENCES

1. Andersen, N.E, Kensing, F., Lundin, J., Mathiassen, L., Munk-Madsen, L., Rasbech, M., and Sørgaard, P.

Professional System Development: Experience, ideas and action. Prentice Hall, New York, 1990.

2. *BYTE Magazine*. Special issue on ComponentWare. 19, 5 (May 1994).

3. Cameron, D., and Rosenblatt, B. *Learning GNU Emacs*. O'Reilly & Associates, Sebastopol, CA, 1991.

4. Cox, B. J. Planning the Software Industrial Revolution. *IEEE Software*. 7, 6 (Nov. 1990), 25-33.

5. Cypher, A. Customizing Application Programs, in *Proc. First International HCI'91 Workshop* (Moscow, Aug. 5-9, 1991), pp. 152-157.

6. Dourish, P. Accounting for System Behaviour: Representation, Reflection and Resourceful Action. In this volume. 1996.

7. IEEE Std. 610.12-1990. Glossary of Software Engineering Standards Collection, *IEEE CS Press*, 1993.

8. Eisenberg, M Programmable Applications: Interpreter Meets Interface. *SIGCHI Bulletin*. 27, 2 (April 1995), 68-93.

9. Fischer, G., and Girgensohn, A. End-user Modifiability in Design Environments, in *Proc. CHI'90 Human Factors in Computing Systems* (Seattle, April 1-5, 1990), ACM Press, pp. 183-191.

10 Fischer, G., McCall, R., and Morch, A. Janus: Integrating hypertext with a knowledge-based design environment. *Proc. Hypertext'89* (Pittsburgh, November 5-8, 1989), ACM Press, pp. 105-117.

11 Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. Seeding, Evolutionary Growth, and Reseeding: Supporting the Incremental Development of Design Environments. *Proc. CHI'94 Human Factors in Computing Systems* (Boston, April 24-28, 1994), ACM Press, pp. 292-298.

12 Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., and Pintado, X. Class Management for Software Communities. *CACM*. 33, 9 (Sept. 1990), 90-103.

13 Girgensohn, A., Zimmermann, B., Lee, A., Burns, B., and Atwood, M.E. Dynamic Forms: An Enhanced Interaction Abstraction based on Forms. *Proc. INTERACT'95* (Lillehammer, June 25-29, 1995). Chapman & Hall, London, pp. 362-367.

14 Grønbæk, K., Kyng, M., and Mogensen, P. Cooperative Experimental System Development: Cooperative Techniques Beyond Initial Design and Analysis. In this volume. 1996.

15 Henderson, A., and Kyng, M. There's no place like home: Continuing design in use, in *Design at Work* (Greenbaum, J & Kyng, M., eds.). Lawrence Erlbaum, Hillsdale, NJ. 1991, 219-240.

16 Hutchins, E.L., Hollan, J.D., and Norman, D.A. Direct Manipulation Interfaces. in Norman, D.A. and Draper, S.W. (Eds.) *User-Centered System Design*. Lawrence Erlbaum, Hillsdale, NJ, 1986, 377-398.

17 Hölzle, U. Integrating Independently-Developed Components in Object-Oriented Languages, in Nierstrasz, O.(ed) *Proc. ECOOP '93*, LNCS 707, Springer-Verlag, Kaiserslautern, 1993, pp. 36-56.

18 Mackay, W.E. Triggers and Barriers to Customizing Software. *Proc. CHI'91 Human Factors in Computing Systems* (New Orleans, Apr. 27-May 2, 1991), ACM Press, pp. 153-160.

19 MacLean, A., Carter, K., Lovstrand, L., and Moran, T. User-tailorable Systems: Pressing the issues with Buttons. *Proc. CHI'90 Human Factors in Computing Systems* (Seattle, April 1-5, 1990), ACM Press, pp. 175-182.

20 Madsen, O.L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Wokingham, 1993.

21 Malhotra, J. Dynamic Extensibility in a Statically-compiled Object-oriented Language. *Proc. of the International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, Nov. 1993.

22 Malhotra, J. On the Implementation of an Interpreter for Building Extensible Applications. Technical report, Computer Science Department, Aarhus University, Oct. 1993.

23 Malone, T.W., Lai, K-Y., and Fry, C. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. *Proc. CSCW'92* (Toronto, Oct. 31-Nov. 4, 1992), ACM Press, pp. 289-297.

24 Moran, T. P., and Carroll, J.M (eds.). *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum, Hillsdale, NJ, 1996.

25 Mørch, A. Application Units: Basic Building Blocks of Tailorable Applications. *Proc. 5th International East-West Conf. on Human-Computer Interaction*, (Moscow, July 4-8, 1995). (LNCS 1015). Springer-Verlag, Berlin, 1995, pp. 45-62.

26 Mørch, A. Designing for Radical Tailorability: Coupling Artifact and Rationale. *Knowledge-Based Systems*. 7, 4 (Dec. 1994), 253-264.

27 Nardi, B. *A Small Matter of Programming: Perspectives on End User Computing.* The MIT Press, Cambridge, MA, 1993.

28 Oppermann, R. (ed.). *Adaptive User Support.* Lawrence Erlbaum, Hillsdale, NJ, 1994.

29 Schneider, D. *The Tao of AppleScript*, 2nd Edition. Hayden Books, Berkeley, CA, 1994.

30 Sumner, T., and Stolze, M. Evolution, not Revolution: PD in the Toolbelt Era. In this volume. 1996.

31 Trigg, R.H., and Bødker, S. From Implementation to Design: Tailoring and the Emergence of Systematization in CSCW. *Proc. CSCW'94* (Chapel Hill, Oct. 22-26, 1994), ACM Press, pp. 45-54.

32 Trigg, R.H., Moran, T.P., and Halasz, F.G. Adaptibility and Tailorability in NoteCards. *Proc. INTERACT'87* (Stuttgart, 1987), Elsevier Science Publishers, pp. 723-728.

33 White, J.E. Mobile agents make a network an open platform for third-party developers. *IEEE Computer.* 27, 11 (Nov. 1994), 89-90.