# Tailorable Systems: Design, Support, Techniques, and Applications

**Jawahar Malhotra**

**Ph.D. Thesis**
Department of Computer Science, Aarhus University
Aarhus, January 1994

Department of Computer Science
Aarhus University
Ny Munkegade 116
DK-8000 Aarhus C, Denmark

E-mail: malhotra@daimi.aau.dk

*To Dad*
*here's another one to add to your list*

# Abstract

A tailorable system is one that can be tailored in its use-environment, without any changes to the source-code of the original system. Such a system must allow its users to make significant changes to its functionality, but without any modifications to its source-code. One way to accomplish this is to write the system in a manner such that changes to the system's functionality can be made by *extensions* to its source-code as opposed to *modifications* of its source-code. A system written in this manner is an *extensible* system.

The goal of this dissertation is to study the problems encountered in the process of developing highly extensible systems, and in the process of tailoring them. The study is logically divided into four major parts: (1) *design* deals with issues in the design of extensible systems, (2) *support* explores the language-level and compiler-level support necessary for developing extensible systems, (3) *techniques* illustrates some tested techniques for developing extensible systems, and (4) *applications* deals with the application of the other three parts to create tailorable applications in specific domains.

The main contributions of this dissertation include: (1) an approach for implementing tailorable systems based on embedding an interpreter into a framework-instance with open points, (2) a technique for making interpreted objects persistent, (3) identification of language mechanisms which are suitable for writing tailorable systems, (4) an investigation of the relationship between object-oriented programming concepts and extensibility, (5) dynamic extensibility in Beta, and, in general, in any static object-oriented language, (6) a tailorable hypermedia system which allows source-level tailoring in order to define new media-types, (7) a technique for making a batch-oriented direct-manipulation-based user-interface generator interactive, (8) techniques for the implementation of an interpreter for an object-oriented language like Beta, (9) a comprehensive overview of the area of tailorable systems, and (10) many large working systems like the Beta interpreter, and the tailorable hypermedia system.

The dissertation comprises of five, independently written, related papers. "On the Construction of Extensible Systems" presents an approach for the construction of extensible systems and focuses on the language mechanisms which allow extensible systems to be constructed. "Dynamic Extensibility in a Statically-compiled Object-oriented Language"

presents a technique for introducing dynamic extensibility in Beta; as part of this technique it presents a Beta-interpreter library with its API. "On the Implementation of an Interpreter for Building Extensible Applications" presents the implementation details of the Beta interpreter, discussing only the interesting issues. "Extensibility as the basis for Incremental Application Generation" presents a technique for using the interpreter to transform a user-interface generator, which generates code in Beta, from a batch-oriented system into an interactive system. "Building Tailorable Hypermedia Systems: the embedded-interpreter approach" presents an approach for building tailorable systems in the domain of hypermedia systems.

# Acknowledgements

This work was carried out within the DeVise project at Aarhus University. I would like to thank Ole Lehrmann Madsen and Kaj Grønbæk for the support they offered during this work. They have been a source of ideas as well as encouragement. My knowledge of all the intricacies of Beta can be attributed to Ole. Thanks also to Randy Trigg for introducing me to this field of tailorable systems. His insightful seminar on tailorable systems laid the foundations for much of my research work.

I am grateful to Kurt Jensen for giving me this opportunity to come to Aarhus and to Robert Shapiro for proposing this idea. Thanks to all the members of the DeVise project for making this a fun and exciting place to work. Søren Christensen and Niels Damgaard Hansen helped me get started here in Aarhus; thanks to them. Jørgen Nørgaard helped me on numerous occasions with the two new languages I was learning: Beta and Danish. Kjeld Høyer Mortensen helped review some of this dissertation; he also gave me company on those many late nights at Daimi. Elmer Sandvad helped me with many aspects of this work. Jørgen Lindskov Knudsen, Morten Kyng, Henry Michael Lassen, Søren Brandt, Lennert Sloth, and Jens Arnold Hem helped in various stages of this work. Angelika Paysen and Karen Kjær Møller helped with various administrative issues. I would also like to thank Peter Andersen and Kim Jensen Møller of Mjølner Informatics for the special support I was given during my work with the interpreter. Görel Hedin provided me with much necessary feedback on parts of this work. Thanks for reading so many versions of my papers.

I would like to thank my wife Monisha, for being patient on all those lonely occasions when I had to work, and for all the support she provided me. I thank my parents for encouraging me and having faith in my abilities.

This work has been generously supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. Kaj Grønbæk's work on Chapter 6 was supported by the ESPRIT II/III projects EuroCoOp and EuroCODE.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Overview

## 1.1 Introduction

A tailorable system is one that can be tailored in its use-environment, without any changes to the source-code of the original system. Such a system must allow its users to make significant changes to its functionality, but without any modifications to its source-code. One way to accomplish this is to write the system in a manner such that changes to the system's functionality can be made by *extensions* to its source-code as opposed to *modifications* of its source-code. A system written in this manner is an *extensible* system.

The goal of this dissertation is to study the problems encountered in the process of developing highly extensible systems, and in the process of tailoring them. The study is logically divided into four major parts: (1) *design* deals with issues in the design of extensible systems, (2) *support* explores the language-level and compiler-level support necessary for developing extensible systems, (3) *techniques* illustrates some tested techniques for developing extensible systems, and (4) *applications* deals with the application of the other three parts to create tailorable applications in specific domains.

A system can be tailorable statically or dynamically. A statically-tailorable system is more like a development framework, while a dynamically-tailorable application is a ready-to-run application which can be extended during execution. Many of the issues in the construction of statically and dynamically tailorable applications are the same. Although this dissertation touches upon the static issues, the main goal is to develop dynamically tailorable applications.

As examples of tailorable systems consider a spreadsheet system which allows its users to define new types of cells, e.g. video cells, or a hypermedia system which allows its users to define new media-types.

The area of tailorable systems overlaps with many areas of computer science. The primary areas touched upon in this work include object-oriented programming and languages, compiler and interpreter design and implementation, reflection and meta-level architectures, and tool-integration techniques.

The dissertation is organized as a collection of five related papers:

1. On the Construction of Extensible Systems (Chapter 2) (summary: Section 1.5.1).
2. Dynamic Extensibility in a Statically-compiled Object-oriented Language (Chapter 3) (summary: Section 1.5.2).
3. On the Implementation of an Interpreter for Building Extensible Applications (Chapter 4) (summary: Section 1.5.3).
4. Extensibility as the basis for Incremental Application Generation (Chapter 5) (summary: Section 1.5.4).
5. Building Tailorable Hypermedia Systems: the embedded-interpreter approach (Chapter 6) (summary: Section 1.5.5).

All the papers have a common underlying theme, but are otherwise written independently. They share common terminology and discuss related issues. The papers appear here in almost their original form. The bibliographies from the individual papers have been merged into one common bibliography for the entire dissertation. In addition, the primer on Beta has been removed from the papers which had it, and now appears as an appendix (Appendix A) of the dissertation.

Figure 1.1 presents an overview of all the papers: for each paper, the area of focus is illustrated. Overlapping areas indicate some common material covered by the papers. (1) presents an approach for the construction of extensible systems and focuses on the language mechanisms which allow extensible systems to be constructed. (2) presents a technique for introducing dynamic extensibility in Beta; as part of this technique it presents a Beta-interpreter library with its API. (3) presents the implementation details of the Beta interpreter, discussing only the interesting issues. (4) presents a technique for using the interpreter to transform a user-interface generator, which generates code in Beta, from a batch-oriented system into an interactive system. (5) presents an approach for building tailorable systems in the domain of hypermedia systems; unlike the abstract discussion in (1), this discussion is based on a real experiment: a large tailorable hypermedia system.

This overview paper begins with some motivation for studying the area of tailorable systems. It identifies some of the interesting problems in building tailorable systems.

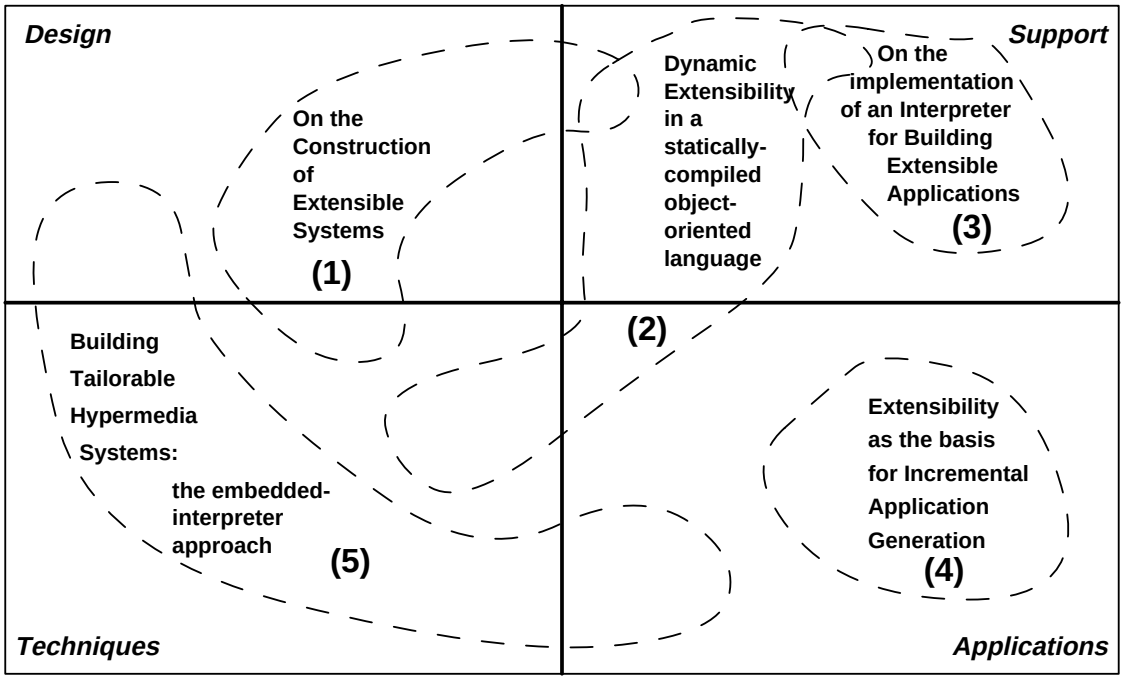


Figure 1.1: An overview of the papers

It summarizes the main contributions of the dissertation as a whole. It also contains a summary of each of the papers in the dissertation. In addition to just describing the papers, it attempts to bring out the relationships between them; the overlaps between material in various papers are also identified. Descriptions of the various working systems built as part of this dissertation-work, and not contained in one of the other papers, are also presented. This paper also quantifies the software-development effort that underlies the dissertation, by listing the sizes of the various systems built. The paper includes an extensive section on related work; the coverage here has greater breadth, and in some cases depth, than the related-work sections in any of the individual papers. Conclusions drawn from the dissertation as a whole are presented. The paper ends with an identification of future directions for the work presented.

## 1.2 Why tailorable systems?

Recently, there has been a trend in the software-development industry to produce highly tailorable systems: Microsoft Word and Microsoft Excel are examples of highly customizable systems. This is in spite of the fact that producing a tailorable system is more expensive than producing a non-tailorable one. It is driven by the realization that a finished non-tailorable software system cannot possibly meet the needs of all users and use-organizations. The system must be tailorable in order to be able to fit the user's

requirements.

The need for tailorable systems is also well established by many researchers. In [Nyg89], Nygaard, in his article on Profession Oriented Languages, claims:

> *It is commonly argued that users in the future should be given tools for modifying and extending information systems.*

In [BBE90], the authors present some of the problems in the enhancement of the Customer Service System in a bank. They claim the following:

> *We believe that many problems are connected to the attitude that computer systems are finished products.*

Suchman and Trigg, in [ST91], claim that:

> *... effective design involves a co-evolution of artifacts with practice. Where artifacts can be designed by their users, this development goes on over the course of their use.*

Henderson and Kyng, in [HK91], claim that one of the goals of their work is to contribute to the creation of systems that better fit work situations, as well as the aims and intentions of the people using the systems.

> *This implies a view of design as a process that is tightly coupled to use and that continues during the use of a system.*

There is even an argument in the context of groupware [Gre91]:

> *For groupware to be considered successful, it must be usable and acceptable by most, if not all, members of the group. ... the notion of personalizable groupware is proposed, defined as a system whose behavior can be altered to match the particular needs of group participants and of each group as a whole.*

In [TMH87], the authors state:

> *The need for adaptable systems stems from a common complaint of users that their systems don't "fit" either the particular task they are doing, their style of working, or their personal sense of aesthetics. This situation leads to one of several outcomes: (1) the users live with the unsatisfactory state of affairs (or*

*stop using the system), (2) the system is modified to meet the new requirements (involving, say, a minor "tweak" at the user site, another iteration in a rapid prototyping sequence, or a massive redesign and re-implementation), or (3) the user produces the new system behavior without help from programmers or designers. Though there are certainly cases when (1) or (2) are desirable, we feel that systems designed to support option (3) hold the greatest promise for success in the long run.*

The underlying message in all of these statements is the same: in order for a system to be more effective and usable, it must be tailorable in its use environment.

There are additional arguments for building tailorable systems. A tailorable system acts as a catalyst for new ideas: advanced users can easily experiment with various extensions, leading to some very creative uses of the system, not conceived of at development time. GNU Emacs [Sta84] is a good example of this. A tailorable system also offers better possibilities for integration with other systems. Certain types of tailorable systems, like the Silica window system [Rao91], or the Common Lisp Object System (CLOS) with the Meta Object Protocol (MOP) [KdRB91], also allow the user to make certain parts of the original implementation more efficient. In Silica, for example, a user can build a spreadsheet system by using a window for each cell in the spreadsheet. Normally the overhead of doing this would be prohibitive, but in Silica, once can tailor the window concept for use in a spreadsheet, thus eliminating this overhead.

## 1.3 Challenges in Developing Tailorable Systems

This section summarizes the interesting challenges in the development of tailorable systems.

System development typically comprises of many phases: e.g. requirements analysis, modeling, design, implementation, use. In building a tailorable system, one can focus on each of these phases to determine what new issues must be considered. For example, in the requirements-analysis phase, one typically captures the current requirements of the user. In order to create the requirements for a tailorable system, however, it is also necessary to capture domains of anticipated changes in the system. If the system is to be used by a group of users, it is impossible for them all to be comfortable with the same set of requirements. It is inevitable that there will be differences in how the system will be used by different users. These differences must be captured as much as possible. In addition, foreseen organizational changes should be taken into account. In short, any circumstances which could affect the use of the system must be captured. Hence, instead of ending up

with requirements for a single system, one should end up with requirements for a range of systems. An interesting research problem is to try to formalize the analysis phase for a tailorable system. [Tri93] explores this line of thought in considerable detail.

The modeling phase can also be extended to handle tailorable systems. Here, the initial model, produced using conventional techniques, must go through a process of generalization. Instead of just modeling the specified system, the model should capture the problem domain in general. This process will lead to insights into which aspects of the system should be tailorable. Such a modeling exercise is described in [KN93]. Another interesting research problem would be to study the modeling phase for tailorable systems.

Analysis and modeling, however, haven't been the subject of this dissertation. Although they have been examined in some detail, there still remains much to be done in those areas. An experimental approach, in which these phases are conducted for a variety of real tailorable systems, is probably the best one to adopt.

The focus of this dissertation is primarily on the design, and implementation phases, with some discussions on the use phase. In the design phase, the model is transformed into an implementation architecture. A number of decisions are made during this phase. These influence the tailorability of the resulting system. Taking the example of a hypermedia system, the model of the system may describe the structure of a media-type and the various operations on it. It doesn't say anything about how the implementation of a media-type should be organized. There are endless choices to be made. Tailorability concerns can constrain this set of choices. The hypermedia system presented in Chapter 6 implements a media-type in many different layers; this is primarily to make the system highly tailorable, as well as easily portable. This is one of the problems addressed by this dissertation.

In the implementation phase, the architecture must be realized in some programming language. The types of tailorability described by the requirements, the model, and the architecture, must be realizable in the chosen programming language. This dissertation examines language-level issues; it captures what exactly is needed, in terms of a language, in order to implement tailorable architectures.

Most dynamically-tailorable systems today are of two types: (1) developed in a dynamic language, with tailoring supported by the language and its environment, and (2) developed in a static language with tailoring supported by an embedded interpreter/compiler for some static or dynamic scripting language. Notecards [TMH87] is an example of (1). Microsoft Excel [Mic93], ACE [JNZM93], Emacs [Sta84] are examples of (2). For reasons discussed in Chapter 2, and justified time and again, this dissertation deals with systems of type (2). However, instead of using the approach in which a scripting language, different from the original development language, is used, it chooses to use the original language

as the scripting language. This, in addition to being potentially more powerful than some ad hoc scripting language, also allows direct access to the internals of the original system. In addition, as is shown in the thesis, it makes it next to trivial to introduce dynamic tailorability into a system which is designed to be tailorable.

This does, however, create new problems to be dealt with. Since the original development language is a static one, a scheme must be devised for handling dynamic extensions. In fact, the language chosen by the dissertation is the statically-compiled, object-oriented, block-structured, strongly- and statically-typed language Beta [MMPN93]; Chapter 2 motivates this choice of language in great detail. In order for a system to accept Beta as the "scripting" language, it must be able to compile or interpret Beta code in its own context. The problem then is to build stand-alone systems, written in Beta, which can accept extensions, written in Beta, at run-time. The problem is made more difficult because of the fact that Beta is very static in nature.

Beta applications often use a persistent store or an object-oriented database (OODB) to store object instances. In general the classes describing these object instances are compiled into the application saving or loading the objects. Once the application is extensible, new classes, and hence new object instances, which are not compiled into the application, can exist at run-time. A problem arises when these object instances are saved into the persistent store or OODB, for, if later, they are loaded by an application which has not been extended appropriately, the results will be unexpected.

The use phase is when the user tailors the system. There are many problems that can be examined here. The most interesting ones deals with comprehension of the system in order to tailor it. This is not dealt with directly in this dissertation. Other researchers are working on this area (see Section 1.6). This dissertation does illustrate the type of effort required of the user in order to tailor a hypermedia system. Also addressed is the issue of building domain-specific tailoring tools in order to simplify the task of the tailoring user.

Today, a number of tailorable systems exist. Some of these applications offer *shallow* tailoring (e.g. Microsoft Excel), where macros can be built, or the user-interface can be reorganized. Others offer *deep* tailoring (e.g. Emacs [Sta84], Notecards [TMH87]), where significant changes to the original functionality can be made. There are, however, only a few published systematic approaches to building tailorable systems. In the reflection community, there is work on building meta-level architectures ([KdRB91, CM93]); the focus of this work is however the domain of programming languages and their implementation, although [CM93] uses reflection in order to implement distribution. The problem of presenting a systematic approach for building a tailorable system is addressed by this dissertation: it describes an approach for building tailorable hypermedia systems.

## 1.4 Main Contributions

This section summarizes the main contributions of this thesis.

1. An approach for implementing tailorable systems. This approach can be stated simply as "framework-instance + interpreter + open-points = tailorable system."
2. A technique for making interpreted objects persistent.
3. Identification of language mechanisms which are suitable for developing tailorable systems.
4. An investigation of the relationship between object-oriented programming concepts and extensibility.
5. Dynamic extensibility in Beta, and, in general, in any static object-oriented language. The thesis shows how to write a Beta application which is dynamically extensible in the sense of Smalltalk, but without all the overhead of having the development environment.
6. A tailorable hypermedia system which allows source-level tailoring of the type found in dynamic environments like Smalltalk and Lisp.
7. A technique for making a batch-oriented direct-manipulation-based user-interface generator interactive. This technique is based on extensibility and the incremental generation of source-code.
8. Techniques for the implementation of an interpreter for an object-oriented language like Beta. In particular, the ability to load an interpreted class into the execution of a compiled program, and allow the interpreted class access to symbols, compiled or interpreted, defined in its scope.
9. Comprehensive overview of the area of tailorable systems.
10. Many large working systems: an interpreter for Beta, an interactive Beta environment, a tailorable hypermedia system. The total development effort amounts to approximately 15,000 lines of Beta source-code.

## 1.5 Summaries of the papers and other work

This section summarizes each of the papers contained in the dissertation. It relates them to each other and also describes work not contained in any of the papers.

### 1.5.1 On the Construction of Extensible Systems

This paper, included as Chapter 2, presents an approach for the construction of highly extensible systems. The approach is based on constructing the system so that it architecturally resembles a piece of computer hardware. Language mechanisms that enable and encourage the construction of systems in this style are identified and discussed. The paper demonstrates, by an example, the construction of an extensible system. It also examines the relationship between object-oriented languages and extensibility. It does this by viewing extensions with respect to the inheritance hierarchy of a system, and examines the types of changes one can make to the hierarchy by writing and installing extensions.

The chapter uses, as an example, a spreadsheet system which allows its users to specialize existing cell-types and introduce new cell-types. The system is dynamically tailorable, and the tailoring process does not require the original source-code or development environment. This example is similar to the one used in Chapter 3 (summary: Section 1.5.2). There, the example is used to illustrate the API of the interpreter, while here the focus is on the design approach and the implementation techniques for constructing an extensible application. This chapter discusses the component-level architecture of the system. The interpreter, to be completely described in Chapter 3, is used here to show how a system can be made dynamically tailorable.

The chapter shows the architecture of the spreadsheet system in Beta. The construction is done in the hardware style. The construction illustrates the language mechanisms that enable and encourage this style of construction. It illustrates techniques for implementing an extensible system. The abstract code of a statically-extensible spreadsheet system is presented. In this system, text cells and picture cells are supported, and the behavior of both can be extended without any changes to the original system.

The various language mechanisms (block structure, virtual patterns, further binding of virtuals, inner construct, nested specialization, patterns as first-class values) are discussed in the context of their role in implementing an extensible system. The flavor of extensibility is different from that of, say, Lisp where most of the original system can be changed by extensions. Here, the idea is that of controlled extensibility: allow the user to extend things, but in a controlled and regulated manner.

The use of pattern variables and patterns as first-class values is explored to introduce dynamic extensibility into this spreadsheet system. The construction of a spreadsheet system, in which the text and picture cell-types can be dynamically tailored, and new cell-types dynamically introduced, is shown. The introduction of a video cell to this system is illustrated.

The chapter also explores how such a hardware-style construction would work in another

language like Smalltalk or C++.

Various alternatives for processing extensions to a system dynamically, in the context of the executing system, are explored: (1) an ordinary non-incremental compiler with a traditional linker/loader, (2) an incremental compiler, and (3) an embedded interpreter.

The relationship between object-oriented language concepts and extensibility is explored. To do this, the different types of changes one can make to a system are classified. For each class of changes, it is examined if they can be accommodated by making extensions to the system. This study is presented in a general language-independent manner.

The primary contributions of this chapter are in the area of Design and Techniques, with minor contributions in the Support area.

### 1.5.2 Dynamic Extensibility in a Statically-compiled Object-oriented Language

The goal of this paper, included as Chapter 3, is to show how one can have dynamic extensibility in a language, without having all the safety and efficiency problems that normally accompany it. This is accomplished by introducing dynamic extensibility in Beta. A technique for doing this is presented. A part of this technique relies on an embeddable interpreter for Beta. The API for this interpreter is presented, and its use illustrated. It is shown that, using this approach, one can still write safe and efficient Beta programs, only now the programs are dynamically extensible. A couple of additional applications of the Beta interpreter are also presented: an interactive Beta environment, and its use in a source-level debugger. Ideas on supporting dynamic extensibility in other static languages are also presented.

This chapter also uses the spreadsheet example which is similar to the one presented in Chapter 2 (summary: Section 1.5.1). It presents a technique for introducing dynamic extensibility into this spreadsheet system. The technique is based on embedding the Beta interpreter into the application. Interpreted code then runs in the context of the extensible application. There is no need for the source-code of the original application, or the original development environment. The API for the interpreter is designed to allow easy embedding of the interpreter into the application, and to allow extensions in arbitrary lexical scopes of the program being extended. The details of how each of the API functions work is presented. The details of the incorporation of an interpreted extension pattern into a running program are also presented.

The chapter touches upon a few implementation issues like type-checking of the extension and the access of compiled object-code from the interpreted code. It illustrates how

objects which are partly compiled and partly interpreted come into existence and how they are implemented. The treatment here is quite superficial; the paper in Chapter 4 (summary: Section 1.5.3) presents the implementation of the interpreter in greater detail.

The paper also presents some additional applications — in addition to dynamic extensibility — of the interpreter. An interactive development environment with a read-eval-print loop, as in Lisp systems, has been built for Beta. Another application of the interpreter is in the source-level debugger for Beta. Here, it is proposed that the debugger support the execution of arbitrary Beta code at break-points by invoking the interpreter to evaluate the code in the context of the break-point.

The feasibility of this approach to introduce dynamic extensibility in other languages is explored. The set of requirements imposed on a language and its implementation are identified. It is then examined if languages like Simula, Eiffel, and C++, and their respective representative implementations, satisfy these requirements.

Statistics of the interpreter's performance and their comparison with the performance of compiled code are also presented. The overhead of embedding the interpreter into various Beta applications is presented. The results presented in the paper haven't changed much since the writing of the paper, as there hasn't been any effort on profiling the interpreter code and improving its performance.

The primary contributions of this paper are in the Support area. There is also a little mention of Applications and Techniques.

### 1.5.3 On the Implementation of an Interpreter for Building Extensible Applications

This paper, presented in Chapter 4, describes the implementation of the interpreter; its contribution is primarily in the support of extensible systems. The interpreter's API is first mentioned in Chapter 2 (summary: Section 1.5.1) and is elaborated upon in Chapter 3 (summary: Section 1.5.2). This chapter picks up where Chapter 3 leaves off and proceeds to describe, in detail, how the interpreter features, used there, are implemented. The focus of the presentation is on aspects of the implementation that involve the following:

1. handling a static object-oriented language.
2. embedding the interpreter into any Beta application and, thereby, allowing extensions access to the application's internals.
3. specification of interpretation-context in calls to the interpreter.

4. intermixing of compiled and interpreted code.

5. working with a statically- and strongly-typed language, and ensuring the continued type-safety of the extended application.

6. returning patterns as closure objects.

These also happen to be the novel aspects of the implementation. This is *not* an attempt to describe the entire implementation of the interpreter.

This chapter uses, as an example, a graphical editor to illustrate the types of extensions the interpreter is designed to support. It sets up the stage with a scenario in which the graphical editor with an ordinary window is extended, via the interpreter, to an editor with a color window. It then illustrates the interpreter implementation using this scenario.

It shows exactly what happens when the color-window code is loaded by the interpreter. In particular, it shows:

1. The process of constructing the pattern closure for the color-window extension pattern. This process is presented in various stages:

   (a) The installation of the extension into the specified scope of the application being extended is described.

   (b) The type-checking of the extension and the construction of a symbol-table for it are described. It is shown how the symbols used in the definition of the extension are resolved into runtime entities in the executing application. Pattern names get resolved into addresses of prototypes, variable names get resolved into addresses of locations they refer to.

   (c) Building the prototype: the construction of the various parts of the prototype is illustrated. The Inner Dispatch Table, and the Virtual Dispatch Table are focussed. The structure of interpreter-generated object-code stubs is illustrated in a machine-independent manner. These stubs are used to interface interpreted objects with compiled ones.

   (d) Getting the environment object in order to build the closure.

   (e) Building the pattern closure from the prototype and the environment object.

2. Use of the returned pattern closure. It illustrates the creation and execution of a color-window instance.

The chapter also introduces a slight variant of the API function `MakeDeclExecutable` described in Chapter 3. Instead of calling the interpreter with an explicit context argument

and an environment object, the interpreter is invoked with a closure. This, as explained in the chapter, is conceptually cleaner and less error-prone. With the original approach, it was possible for the user to specify incompatible values for the two arguments by passing an environment object which didn't correspond to the given context. With this refined approach, such a situation is impossible.

### 1.5.4 Extensibility as the basis for Incremental Application Generation

This paper, presented in Chapter 5, describes an application of the interpreter. It presents an approach for making a direct-manipulation-based user-interface generator incremental, and hence interactive. It introduces the notion of incremental code-generation.[1] The main concept utilized here is also extensibility of the type described in Chapters 2 and 3. The effect of using this approach is a reduction of the time required between the edit and use phases of a user-interface generator.

The technique is described in the context of a user-interface generator known as the ApplBuilder. The original ApplBuilder is batch-oriented: every time an interface is edited, it regenerates the code for the affected parts of the application; this code must then be compiled and linked before the edited component can be used. With the proposed approach, the ApplBuilder becomes interactive: it generates code incrementally; this code is an extension to the original application; hence, it can be dynamically loaded into the application via the interpreter embedded within the application. There is no need to recompile the application.

To put this in the context of tailorable systems, observe that the ApplBuilder can be viewed as a tailoring tool, and the application whose interface is being edited as the tailorable application. Instead of writing code to tailor the application, the user edits the interface of the application using direct-manipulation, and possibly specifying new code for the behavior of the various user-interface components. The ApplBuilder takes care of doing the source-code-level tailoring. This is a tool for tailoring the user-interface aspects of an application; similar tools can be constructed for other aspects of an application, e.g. for tailoring the media-types of a hypermedia system.

The result of this paper can also be viewed outside the context of tailorable systems. In that case, it is a "compilation" technique which is somewhere in the middle of the continuum between batch compilation and incremental compilation. It gives the effect of incremental compilation, with some execution overhead, but without having all the complexity of incremental compilation.

[1] Not machine code, but code in a high-level language like Beta.

This chapter uses the Find-File dialog from the Macintosh Finder as an example. It illustrates the code generated for this dialog. Then, for each edit operation possible in the ApplBuilder, it shows how the ApplBuilder can generate code for the edited dialog, such that the new code is an extension of the original code. In other words, this newly generated code is such that "original-code + new-code = edited-dialog." Here + denotes the dynamic loading of new-code into original-code. This approach is shown to work even when these edit operations are composed arbitrarily. This proves that any sequence of changes to the dialog can be handled via this approach. The techniques for generating these extensions rely on the ideas presented in Chapter 2.

Also introduced, in this chapter, is the notion of an extension server. This is used by the ApplBuilder to register extensions. The application being edited gets updated by consulting this extension server, and invoking the interpreter embedded within it to load the extension.

The ApplBuilder needs to be adapted in order to use this approach. It now has two modes of code-generation: extension-ready code-generation, and extension code-generation. The extension-ready mode is non-incremental; code for the entire application is generated and needs to be compiled and linked as usual. This code is, however, similar to that of the dynamically-extensible applications presented in Chapters 2 and 3. This is, thus, an example of how such extensible applications can be mechanically generated. The extension code-generation mode is the incremental mode in which extensions to the original application are generated. These modes are discussed in this chapter.

The results of this chapter have been tested by manually generating code using the described techniques. The automation of this approach is forthcoming.

### 1.5.5 Building Tailorable Hypermedia Systems: the embedded-interpreter approach

Chapter 6 presents this paper as the logical conclusion of all of the other papers. Using most of the techniques presented in the other papers, it presents a general approach for constructing tailorable systems in the domain of hypermedia systems.

There are a number of results in this chapter. First, it shows a general technique for implementing a tailorable system: a tailorable system is an instantiation of an object-oriented framework with open-points which can be filled via an embedded interpreter. It presents a hypermedia development framework whose architecture is suited for this type of instantiation into a tailorable system. It presents an implementation technique for making interpreted objects persistent. These are objects that are generated from classes which are

interpreted extensions to the program; thus there is a possibility that the corresponding class may not be defined when such an object is accessed. It presents a specific tailorable hypermedia system, which is source-code-level tailorable, and can be tailored to support arbitrary new media-types. It presents some insight on the survivability of extensions when a new version of a tailorable system is released. Finally, it compares this approach for building tailorable systems with approaches in which the development environment and source-code are available, concluding with overhead measurements that establish the viability of this approach.

The results are presented in the context of a hypermedia development framework known as the DeVise Hypermedia (DHM) framework. The chapter establishes a scenario in which a user of a DHM system wants to include drawings, made with his/her favorite drawing editor, into hypermedia documents made with this DHM system. The DHM system, unfortunately, has support only for text and file components: text components support within-component links, and file components support only whole-component links. Fortunately, the DHM system is tailorable; the user tailors it to use the drawing editor. The user is now able to include drawings in hypermedia documents, with support for within-component linking; i.e. the individual elements of the drawings may serve as anchors for links.

The chapter begins with the above scenario and illustrates the user's interaction with the tailorable hypermedia system in order to tailor it to have the drawing media-type.

The architecture of this tailorable DHM system is described by illustrating, first, the architecture of the underlying framework. The instantiation of this framework into a system, which has open points, and the interpreter embedded within it, is then illustrated. The ease of construction of this tailorable system is emphasized.

The framework architecture comprises of four layers: Application which contains the applications used to manipulate the various media, Presentation which abstracts the applications for the remainder of the DHM system, Runtime which implements the transient behavior of a DHM session, and Storage which implements the storage aspects of the various media into a persistent store or OODB.

The tailoring effort on the part of the user is illustrated by presenting class-diagrams. These show clearly what the user needs to write in order to integrate the drawing editor with the hypermedia system. The class interface of the user's original drawing editor is shown. The drawing editor is specialized to support linking operations which get implemented in terms of the hypermedia system. The hypermedia system's Presentation layer is also specialized to define a new presentation class which can interact with the drawing editor. The simplicity of this code is emphasized.

Since drawings now belong to hypermedia documents, it must be possible to save them into the persistent store, or OODB, as parts of hypermedia documents. A technique to do this safely is illustrated. Should a hypermedia document containing such a drawing component be opened in a hypermedia system without the drawing extension, the extension gets installed automatically.

The chapter also has a brief discussion on the problem that extensions made on one version of a system are often incompatible with a newer version of the system. The main point made, and justified, is that because extensions, written for the types of tailorable systems described here, don't have unrestricted access to the system, they are more likely to survive new versions of the system.

This chapter also presents the tailorability overhead for the hypermedia system. In terms of size, it shows that the tailorable hypermedia system is 2.3 MB larger than the non-tailorable version. This figure includes the increase in size of the executable, and all the additional information necessary for tailoring the executable. This is small, when compared with tailoring environments where the entire development environment and source-code of the original system must be present. More details and comparisons are in Chapter 6.

### 1.5.6 The system-development effort

Underlying this dissertation is a substantial system-development effort. A number of working systems exist at the time of this writing. These include an embeddable Beta interpreter in the form of a library. This reliably supports almost the entire Beta language. An incremental read-eval-print toploop for Beta, which utilizes this library, also exists. This allows interactive and incremental evaluation of Beta programs.

A tailorable drawing editor, used originally for some experimentation, and later for the hypermedia system, exists. A tailorable hypermedia system with support for new media-types is available. The interpreter was also integrated with sif, the syntax-directed editor for Beta programs. In this version of sif, the active window can be interpreted. An integration of the interpreter with a drawing tool called DesignEnv was also completed. Finally, a number of smaller tailorability experiments, which generally comprised of taking an existing Beta application, adding open points to it and embedding the interpreter within it, were also conducted.

The Beta interpreter comprises of 12,101 lines of Beta code, 135 lines of C, and 191 lines of assembly code. The Beta code count does not include the type checker. If the Beta source files are measured using `wc -l`, the total count comes to 17,713. If blank lines are deleted from these files, the count is reduced to 15,126. Assuming that approximately

20% of this is comments, the count is further reduced to the 12,101. The effort involved in the miscellaneous tailorability experiments amounted to 300 lines of Beta code; this figure includes only the code needed to make the systems tailorable, not the original code of the system.

The tailorable hypermedia system amounted to a total of 2,146 lines[2] of Beta code. This does not include the code for the entire hypermedia system; only the code written to make it tailorable, the code for the drawing editor, and the tailoring code for the drawing media-type.

## 1.6 Related Work

There are a number of different areas of work that have something in common with tailorable systems. This section is, therefore, organized by the area of work.

### 1.6.1 Support for dynamic extensibility

In dynamic languages, such as Smalltalk [GR83], Common Lisp [Ste90], CLOS [BDG+89], and Self [US87], the ability to handle dynamic extensions is inherent. In all of these languages, it is possible to write the types of dynamically extensible stand-alone applications we have described here. There is no need for any additional support. In these languages, the compiler is just another object/function which can be invoked to compile some new code dynamically. In addition, due to the dynamic semantics of the language, the runtime system is organized in a way that permits dynamic modification of the executing program. The process of building an application, which is usually done by picking the "root" object/function of the application and asking the system to save an image, automatically includes the compiler, if necessary, in the stand-alone application.

The Smalltalk compiler is just another object which is part of the Smalltalk environment. It may be sent "eval" messages with the source-code to be evaluated. The resulting value, a compiled-method object, can be made a method of some class, by simply inserting it in the appropriate dictionary. If method lookups are cached, it may be necessary to flush the cache. All of this can be done by any Smalltalk program. Common Lisp systems and CLOS implementations have similar capabilities. CLOS, for example, provides access to its interpreter, its compiler, and its binding environment. Using these, any CLOS program can evaluate new code and install it in the environment. Self also provides programs access to its compiler. In Self, it is possible to change the slots of an object, and even change

[2] after deleting blank lines.

its parent(s), all in a running program. There is additional discussion of these issues in Section 3.6.

In our work, we have chosen to work with static languages and introduce dynamic extensibility into it. Static languages offer many benefits such as potentially more efficient programs, strong static type checking, greater program readability, etc. By introducing dynamic extensibility in Beta, we are narrowing the gap between dynamic languages, such as Smalltalk, and static languages, such as Beta. We are getting the dynamic benefits of Smalltalk-like languages without giving up all the benefits of programming in Beta.

Introducing dynamic extensibility into static languages can be done in many different ways. Incremental compilation is a technique which can produce a new executable for a program with a minimal amount of recompilation. [Hed92] contains a thorough discussion of incremental compilation and a semi-automatic scheme for generating incremental compilers for object-oriented languages. When incremental compilation is combined with an execution model which can handle changes to the program during execution, one gets a Smalltalk-like dynamic execution environment. [HM87] presents such an execution model and shows its use in supporting exploratory programming in Simula. The authors illustrate a technique for handling changes to a program during its execution, with the ability to continue execution with the change. So, when there is a change to the program, the incremental compiler is used to recompile only the necessary parts, and the recompiled parts are incorporated into the current execution. They present an execution model which describes the instance world of a program execution. They then illustrate the effect of modifications to the description world, i.e. the program description, on the instance world. They handle issues such as what should happen to existing instances when they are incompatible with the modified program description. Other environments which support correction of errors during program execution are DICE [Fri84], and LOIPE [Fei82]. Although these approaches can be used to construct dynamically extensible systems like the ones we have constructed, they offer more functionality than is necessary for building tailorable systems. They are able to handle arbitrary program *modifications*, while all we are looking to support is arbitrary *extensions*. Sections 2.4 and 3.6 compare our approach with incremental compilation.

There has also been prior work in introducing dynamic extensibility into Beta [AF89]. In this approach, a dynamic linker and loader are implemented and embedded into the extensible system. The Beta compiler, which exists as a separate process, is used to compile extensions. The embedded linker and loader are used to load the compiled extensions. This is described in Section 2.4. More recently, a general purpose dynamic and incremental linker, based on shared libraries, is being made part of the Beta environment [Fre93]. Extensible systems built using this linker will be of the same type as those described in [AF89].

Another widely-adopted approach for building extensible systems is to develop the system in any suitable language, and then embed into the system an interpreter/compiler for a scripting language. The primitives operations of this scripting language provide access to the internals of the system. Users can generally combine these primitives, using the constructs provided by the scripting language, to tailor the system. These scripting languages range from simple, as is the language in Microsoft Excel, to completely general purpose, as is Lisp in GNU Emacs [Sta85]. Another example is the ACE environment [JNZM93] where an interpreter for Scheme [Bet89] is embedded into the system, solely for the purpose of providing tailorability. Such an approach to building extensible systems is not suitable for the type of source-level tailorability we have illustrated in this work. It would be significantly more difficult, though not impossible, to write the types of extensions we write for the hypermedia system (Chapter 6), if all we had was an interpreter for Scheme or Lisp embedded within the system. This would hold even if the hypermedia system had been implemented in Lisp or Scheme; in this case, we would no longer have the extensibility mechanisms of object-oriented languages. Further treatment of some of these issues is in Section 3.6.

Python [vd91] is an object-oriented language with an interpreter written in C. It has access to many modules e.g. window management and UNIX functions. An application can embed this interpreter into itself, thus getting a tailoring language. This would be similar to the scripting-language approach, except that the scripting language would also be object-oriented.

Another unpublished report worth mentioning is one that discusses the dynamic exchange of Beta systems [KMMPN86]. The ideas here are the predecessors of pattern closures as they exist in Beta today.

The architecture of the interpreter built for Beta is similar to that described in [Lie87] in that it comprises of interpretation methods which belong to classes representing the various syntactic constructs in Beta. Hence, every type of node in the Beta Abstract Syntax Tree has an interpretation method.

### 1.6.2 Meta-level architectures and reflection

Systems with meta-level architectures, such as CLOS systems with the metaobject protocol (MOP) [KdRB91], are also highly tailorable. In CLOS with the MOP, it is possible to tailor the semantics of the language. Every class, for example, has a metaobject; as a result, introspection and analysis of the classes in a system is possible. It is even possible to create new classes programmatically. It is possible to introduce specialized class metaobjects, and as a result, support different types of inheritance semantics on a per-

class basis. In fact, it is also possible to define specialized generic-function and method metaobject classes, along with a Beta-like inner construct, to get Beta-like semantics in which superclass methods are executed before methods from subclasses, with control being transferred via the inner construct. [Kic92] contains a motivating discussion on building abstractions that are open. It is argued here that the traditional notion of a black-box is inadequate, and that a meta-level adjustment interface is also necessary for fine-tuning the abstraction. This is presented in a little more detail in Section 4.5.

Another example of the use of a meta-level architecture in a language setting is [MC93]. A kernel implementation for a prototype-based object-oriented language is presented along with a metalevel architecture. Using the metalevel interface, it is possible to extend the kernel to implement the semantics of various prototype based languages such as Self. [CM93] presents a meta-level architecture for C++ programs in which classes and methods may be declared reflective. This is used to implement distribution. For more details see Section 3.6.

Languages aren't the only domain in which meta-level architectures are useful. The Silica window system [Rao91] has a meta-level architecture. Here the notion of implementational reflection, which is a broader view of reflection and can be applied to the design of various kinds of systems, is introduced. It illustrates how a spreadsheet system can be built using windows to represent each cell. This is a natural choice, as the window system already implements much of what is needed in the spreadsheet. Typically, the cost of using a window per cell would be prohibitive. Windows are generally heavy-weight objects with a lot of overhead. Cells are, however, simpler than windows in some respects: they don't overlap, they all share a number of properties, etc. The paper shows how, using metalevel programming, it is possible to create a tailored window system in which the windows are fine-tuned to be cells in a spreadsheet: the storage of the windows (cells) is organized as a 2-d array indexed by location in the grid, the hit-detection function is replaced by one which uses simple arithmetic and array reference, etc.

The Apertos operating system is an object-oriented operating system with a metalevel architecture [Yok93]. Described here is a technique for constructing an object-oriented operating system and its kernel, which divides objects implementing system facilities and applications into two types: base-level objects and metaobjects. This allows various types of tailoring of the operating system e.g. the scheduling and memory management policies.

There is also work on a reflective toolkit for CSCW systems [Dou93a, Dou92, Dou93b]. Illustrated here are examples of situations in CSCW toolkits where metalevel programming would be useful. These example include the locking policy for shared objects, the degree of sharing, and the data distribution policy. The idea is for the toolkit to avoid implementing any one policy and allow the toolkit user to tailor the toolkit, using the

metalevel interface, with the appropriate policies before using it to build actual CSCW systems.

The work in this dissertation is not based directly on creating meta-level architectures. Instead, using the interpreter, the user reflects upon the implementation of the system in order to tailor it. Using a meta-level architecture is orthogonal to using the embedded interpreter approach. In fact, Beta systems with meta-level architectures could have an embedded interpreter. Instead of just tailoring the base-level classes, the interpreter could also be used to tailor the meta-level classes, thus giving the user more tailoring power. The approach for building tailorable hypermedia systems (Chapter 6) does use a kind of metaobject to store the type-information for media-types.

### 1.6.3 On the comprehension of systems for tailoring

In order to tailor a complex system, it is important to be able to understand the system. There is some research in this area. [MY89] describes a representation for design based around a semi-formal notation which allows explicit representation of alternative design options and reasons for choosing among them. This representation is meant to improve the coherence of designs, and hence make it easier for end-users to comprehend the system for tailoring purposes. [Mør93] talks about designs made especially for tailorability. He also advocates coupling the artifact with rationale. He argues that by coupling uninterpreted design rationale (such as pictures, diagrams, stories, usage scenarios, and argumentation) with formal software artifacts, the casual computer user will get enough of background understanding of the artifacts to be able to make meaningful changes to them without understanding the implementation language itself.

[Oss87] discusses a mechanism for specifying the structure of large systems; this could prove useful for gaining an understanding of the system before attempting to tailor it. He introduces the notion of a grid; a grid specification identifies the system layers explicitly, and specifies the system structure and access restrictions in terms of these layers. It emphasizes human readability and uses some novel techniques to make the global structure of large systems clear and visible. Such a specification for a system like the hypermedia (Chapter 6) would be useful; the information used by the end-user to tailor the hypermedia system could be presented in the form of a grid specification.

[FGNR92] presents an integrated domain-oriented design environment in which human-computer cooperative problem-solving tools are embedded into knowledge-based design environments. Such an environment allows designers access to relevant knowledge at each stage of the software development process. They describe knowledge access as the cycle of location, comprehension, and modification. They present techniques to reduce the

cognitive cost in dealing with a huge amount of knowledge. They also present examples of domain-oriented design environments. Although their environments are geared towards system designers and builders, the ideas can be applied to construct similar domain-oriented tailorability environments. These could contain knowledge about the tailorable system and the problem domain, and have tools to help with the process of location, comprehension, and modification.

[Joh92] presents a technique for documenting frameworks. The documentation consists of a set of patterns,[3] called a "pattern language." He shows that, for a framework, a set of patterns can be designed to illustrate the purpose of the framework, the techniques for using the framework, and the detailed design of the framework. As tailorable applications have much in similar to frameworks — they are instances of a framework — patterns can be used to describe them as well. The author also concludes that patterns are a good way to describe frameworks as first-time users of a framework will not usually want to know *exactly* how it works, but will only be interested in solving a particular problem. The same argument applies for users — not necessarily first-time — of a tailorable system when they are attempting to tailor the system; they are only interested in the particular feature they want to tailor, and not knowing all the details of the system.

The specialization interface of a class has been the subject of recent research. This is in addition to the use interface which is what the user of the class sees. The specialization interface is that through which the class is extended — by defining new subclasses and appropriate methods. In [KL92], the problem of specifying the specialization interface is addressed. There is a conflict between describing enough of the internal workings of a class so that it can be specialized, and avoiding saying so much that the implementor of the class has no room to work. This conflict is addressed in great detail. A type system for the specialization interface is proposed in [Lam93]. Both of these papers provide valuable techniques for specifying tailoring interfaces which can help in the comprehension of the system being tailored.

### 1.6.4 Examples of tailorable systems

GNU Emacs [Sta84, Sta85] is a highly tailorable text editor. It is written in C and has, embedded within it, a lisp interpreter. The interpreter has access to many of the C-level functions. The editor can be tailored in many ways. First, new lisp functions can be written and defined as user-invocable commands. The bindings for the various input devices (keyboard, pointer) can be changed by the user. Furthermore, existing commands can be modified by attaching user-defined functions onto the provided hooks

[3] not to be confused with Beta patterns.

(or callbacks). GNU Emacs has been highly customized, and today, numerous packages, which customize it in various ways (e.g. language-specific modes, mail, news), exist.

Commercially, there are systems like Microsoft Excel [Mic93] and Microsoft Word [Mic91]. In Excel, for example, it is possible to construct macros (commands) which are written in a simple scripting language, and in terms of its primitive operations. The menu-bar and the tool-bar are also highly customizable. Furthermore, various highly customized spreadsheets can be created by defining various constraints between the cells. This process, however, can be viewed as just using the system rather than tailoring it. Word offers similar capabilities in that its entire menu-bar and tool-bar can be customized. In addition, these systems also have a notion of an "add-on" by which, other independently development sub-components — like a dictionary, or a bibliography manager — can be loaded into the system, even during execution.

Hypercard [Goo87] can be viewed in two equally justifiable ways: as an end-user programming environment, or as a tailorable stack manager. It is an end-user programming environment as users construct various applications with it, applications which may or may not be based on cards. It is a tailorable stack manager for it provides a generic stack concept and allows users to customize this stack in a variety of ways. Hypercard uses a simple scripting language known as Hypertalk [App88]. This language allows one to define the behavior of various aspects of a stack and the cards that comprise the stack. It is an example of a domain-specific end-user language, specialized to handle cards and stacks.

Applescript [App92] is a general scripting language available on the Macintosh. The primitives of this language provide access to various scriptable applications on the Macintosh. The language resembles Hypertalk in many ways. As an example, a script to ask Excel to open a specified worksheet, select a range of cells, and plot a chart with them, can be written. When such a script is evaluated, in possibly another application, appropriate apple-events are generated and sent to Excel, which must process them and execute the specified commands. In this way the user can tailor his/her workspace. In some scriptable applications, every menu-item is implemented by an Apple-script. This script may be edited by the end-user, thus offering the user some potential to tailor the application itself.

In the Unix environment, there are a number of tailorable applications. The various shells, available under Unix, have configuration files which allow customization through the setting of variables and the definition of new commands. The X Window system can be tailored in a variety of ways: from choosing window managers to setting the defaults for various applications. Even X servers can be extended to support additional features. [Rya90] contains a survey of various scripting languages.

[ZC92, JNZM93] presents a framework for interactive, extensible, information-intensive applications. A part of this framework is an extension language. The unit of programming in this language is like a formula. These formulas are like the types found in spreadsheets, only they are generalized. This is because their applications support other visual formalisms in addition to spreadsheet-like grids of cells. The extension language is processed by an embedded Scheme interpreter. This interpreter is able to call new C++ functions without itself being modified or even recompiled. In this way the end-user gets access to the internals of the original application.

A text editor with support for extensible objects floating in text is presented in [Szy92]. In this model, a text is a sequence of elements. Each such element is either a normal character, or an instance of some extension class. Implementing new classes is the primary way of extending the editor. This simple model of extensibility offers the tailor a number of possibilities.

[MCLM90] describes a tailorable system based around the use of distributed on-screen buttons. Buttons are screen objects in Xerox Lisp which look "pressable." When pressed they carry out an action. They can be used without any understanding of the details of the encapsulated action, and thus are a convenient way to tailor the Xerox Lisp environment for individual user needs. The paper discusses various tailoring techniques centered around buttons. Their claim is that this with this approach, they were able to enable non-programmers to tailor their own workstation environment. This, they claim, is because they have a tailoring architecture which supports a large number of tailoring techniques at differing levels of complexity.

Aquanet [MHRJ91] is a hypertext tool with a goal to provide users with the ability to customize knowledge structures for their specific task. This is accomplished through schemas; every session is controlled by a schema that defines a set of allowable basic objects and relation types. It defines the nature and organization of the knowledge structure the user can construct.

SHARE [Gre91] is a "policy-free" view-sharing system whose kernel supports primitives upon which one can build a broad range of policies to manage floor control. View-sharing software allows any unaltered single-user application to be brought into a meeting. In such a system, it is important that it be possible to switch between different policies depending on user preferences and the operating environment. SHARE supports this by providing an extensive library of floor-control policies for groups to choose from. The paper claims that programming the policies proved easy and quick, but it doesn't say if this programming is meant to be done by the end-users.

The Intermedia system [Mey86] is discussed in some detail in Chapter 6. It is a framework for building hypermedia applications. Notecards [TMH87] is also presented in some

detail in the same chapter. It provides the user with a semantic network of electronic notecards interconnected by typed links. The system's goal is to be adaptable; it does this (1) through being flexible: this stems from the multiple interpretations of individual cards and links depending on the context; (2) through being parameterized: for example, one parameter determines whether Browsers distinguish link types by different styles of dashing or whether all link types are drawn the same; (3) though being integratable: for example, it has been used to provide access to remote databases and also been interfaced to a mail system; (4) through being tailorable: for example, users may create new functionality by using the programmer's interface, which can allow users to modify the characteristics of individual cards and links as well as for building new structures.

This dissertation presents techniques for building tailorable applications like these and more. The focus of this dissertation has not been so much on inventing scripting languages as it has been on presenting general techniques for building tailorable systems. Once a tailorable architecture for an application is designed, the scripting language can be easily constructed as a specialization of Beta. This dissertation also shows how to build applications which allow source-level tailoring, as in Notecards and Intermedia, but with a smaller overhead.

### 1.6.5 Language-level issues in developing extensible systems

Although the title of the section is generally applicable to any language, most of the discussion here is about object-oriented languages.

[NS90] presents a discussion of tailorability in Beta. This is also discussed in some detail in Section 2.6. [Nør92] defines the notion of an open point and shows how open points can be implemented in various languages. An open point is a behavioral parameter of an executable program. The paper elaborates on the hook mechanism in Lisp environments and proposes a set of tools which support the filling of these open points. Our work has also concentrated on developing applications with open points, an open point in Beta being a pattern variable. The hypermedia system in Chapter 6 has open points which get filled by the interpreter. Section 2.6 has more details on this.

A natural way to extend a system is to merge its inheritance hierarchy with another, possibly sparse, inheritance hierarchy, resulting in a new system. Such an approach is presented in [OH92]. The motivation for their work is similar to ours: to extend (or tailor) existing systems. They propose an approach in which extensions of all kinds are clearly separated from the base hierarchy. The entire system is obtained by combining the extension hierarchies with the base hierarchy. Sequences of successive extensions can be combined, parallel extensions can be combined, and the base hierarchy can be replaced

without changing the extension hierarchies. They also outline a technique for implementing such hierarchy combination. Their approach is declarative and hence, conceptually, clean. The extension developer simply writes the sparse hierarchy and then merges it using some high-level merge operations; the implementation takes care of the rest. The tailorable systems described in this dissertation also support some form of hierarchy combination. For example, in the hypermedia system (Chapter 6), the base-hierarchy — which is the hierarchy of the hypermedia system — is being extended by a sparse hierarchy — which is the hierarchy of the drawing media-type — to yield an extended system. Our approach is, however, more imperative and less general. Given the semantics of Beta, and the organization of its runtime system, it would be difficult to support all their hierarchy combinations operations dynamically and efficiently. This is also related to the discussion in Section 2.4. Note also that the Beta compiler, in combination with its fragment system, provides a similar form of hierarchy combination, although it does this statically.

Subject-oriented programming [HO93] is an approach to programming which facilitates the development and evolution of suites of cooperating applications. Applications cooperate both by sharing objects and by jointly contributing to the execution of operations. One of the stated requirements is that unanticipated new applications, including new applications that serve to extend existing applications in unanticipated ways, must be supported. Building a system in a subject-oriented style thus results in a highly extensible (and hence tailorable) system: all objects defined in a subject-oriented manner are freely extensible by any other applications.

Traces [Kic93] is an object-oriented language concept which supports specialization of objects in cases which are ordinarily difficult to handle. The example used is that of a graphics editor in which line segments can be moved and resized freely. An end-programmer extends the system by adding a new kind of line segment, with the specialized behavior that its slope cannot be changed. The graphics editor has an operation which groups a collection of lines to form a polygon. It would be desirable that a polygon formed with a fixed-slope line should not allow rotation, as this will change the slope of the line. How then can the graphics editor be implemented in order to create this specialized polygon, whenever it comes across a fixed-slope line. The idea in the paper is to attach a trace, which is a packet of behavior, to the fixed-slope line object. This trace automatically propagates to the object's progeny, where it installs the appropriate behavior. So, a trace, which restricts the rotation of polygons, is attached to all fixed-slope line objects. When they form a polygon, they affect the polygon object's rotation behavior. This technique is useful in the implementation and tailoring of a tailorable system. Without it, the end-programmer would have to ensure that when they specialize a line into a fixed-slope line, they must also specialize polygon into a non-rotating polygon. In addition, they must specialize the polygon-creation code to create such a specialized polygon if a fixed-

slope line is involved. In the context of the hypermedia system described in Chapter 6 one could imagine attaching a trace to a drawing-component object such that it affected the behavior of the session manager, e.g. modified the saving/loading routines. So, for example, every time a session manager object was created, and a drawing-component was part of it, the session manager object would be specialized.

Much research has also been done on creating tailorable programming environments. Much of this generally boils down to tool-integration problems and techniques. It is however relevant to the area of tailorable systems, as many times, tailoring a system is equivalent to integrating it with another. This is true for the hypermedia system described in Chapter 6. Here, tailoring the system to add the drawing media-type involves integrating the drawing editor into the hypermedia system. Harrison and Ossher address the problem of adding new tools to an integrated set and extending existing tools without invalidating the existing tools [HO90a, HO90c]. They have an approach to object definition which emphasizes the need to facilitate extension without disruption. [HO90b] motivates the need for extensible programming. It, like us, also claims that extension can be achieved without access to source-code. It introduces the notion of extension by addition, and the concept of a subdivided procedure. A subdivided procedure has an interface, a subdivision specification, and one or more bodies. When it is called, its subdivision specification is evaluated and the selected body executed. The technique allows new bodies to be added to an application without any need to recompile the application. Such a technique could be implemented in Beta, or any other object-oriented language. In Beta, for example, a pattern could be declared as subdivided. With its declaration, a subdivision criteria, and a default "body" — which would be an object descriptor — would be provided. Furthermore, it would be possible to add new bodies — descriptors which are extensions of the default descriptor — as additional definitions of this subdivided pattern. Each additional body would have a selection value. Such additional body declarations may appear separate from the subdivided pattern declaration. So, a subdivided pattern would denote a set of object descriptors rather than a single one. At every use of such a subdivided pattern, the subdivision criteria would be evaluated, and the body with the matching selection value selected. Thus, the definition of the pattern would depend on the result of the evaluation. If the language supported such a mechanism, it would be simple to use the interpreter to dynamically add new bodies to a subdivided pattern, thus providing another way to achieve dynamic extensibility.

### 1.6.6 Discussion of tailorability issues

The development of an environment in which designers and end-users work together building systems just as easily as building a model house using Lego bricks is the subject

of [KN93]. They have developed a domain model and an application framework supporting project management in general, and quality management in particular. Their idea is to build a hierarchy of specialized development environments, each of which comprises of a domain model component, and an application framework. Such a domain-specific development environment will be more comprehensible to users with knowledge of the domain. It will be possible to base tailoring and adaptation of these systems on concepts largely inherited from the domain.

It is worth mentioning the work on component-oriented software development [NGT92]. They define application engineering as the activity of abstracting the domain knowledge for selected application domains, developing reusable software components to address these domains, and encapsulating this knowledge into generic application frames (GAFs). Application development is then the activity of instantiating a specific application from a GAF to meet some particular requirements. This, in some ways, is similar to the ideas in [KN93]. A component-oriented software system will be tailorable, at least at the component level.

[Tri93] addresses the relationship between participatory design and tailorability. He argues, among other things, that decisions and tradeoffs encountered in the design process of a tailorable system can be influenced by closer contact with users.

As mentioned in the section motivating the development of tailorable systems, there is much research work that argues for the need for tailorable systems. This work includes [HK91, ST91, BBE90, Nyg89]. [Mac90] presents a study of how users within an organization share customizations of software.

## 1.7 Conclusions

A systematic approach for building tailorable systems has been developed. Issues including the design, the language mechanisms, the support systems, the implementation, and the tailoring process have been covered. It has been demonstrated that dynamically tailorable systems, with support for source-level tailoring, can be built relatively easily using this approach. Furthermore, these tailorable systems are not much more complicated than their non-tailorable counterparts. The introduction of a few well-placed open points is all that is necessary to make the system dynamically tailorable. It has also been shown that the overhead incurred by making applications tailorable using this approach is minimal.

Tailorable systems built using this approach are like specialized domain-oriented programming languages/environments. Taking the hypermedia system (Section 6.4.1) as an example, the domain-oriented language comprises of Beta classes like the instance-

presentation class along with its relevant methods, and the session-manager class along with its methods. One first thinks of tailoring at this level of abstraction. A tailor might think of tailoring the system as follows: (1) Make a specialized presentation class which invokes operations in the specialized drawing editor, (2) make a specialized drawing editor that invokes linking operations in the hypermedia system and provides support for the presentation-class protocol. This is clearly a domain-oriented algorithm: it is expressed in terms of domain-oriented concepts.[4] The benefit of this is clear: a hypermedia system user can understand such an algorithm. In fact, with a little training, which need not involve Beta, he/she could even start creating such algorithms. The crucial point here is that having such a domain-oriented language makes the tailoring aspects comprehensible for the user.

The algorithm above is, however, not sufficient to implement the drawing media-type. It is necessary to move down a level of abstraction and think about the methods of the presentation class and the drawing editor. This is also supported by the language. As an example, the tailor can now think: the hasSelectedObject method of a drawing presentation object should call selectionEmpty in the specialized drawing editor to determine if the editor has a selected object. Moving down another level, one could think of the X-windows level, where code must be written to handle the addition of a new link menu to the drawing editor. Finally, there is the lowest level where one programs in Beta and thinks of just patterns, objects, and variables. The development of the lower levels will most likely be done by a programmer with training in Beta. The central point here is that the tailoring language supports all these levels of programming. It is a hierarchical domain-oriented language.

To conclude this line of thought, such a hierarchical domain-oriented language is crucial to the success of tailorable systems. It is a difficult task to eliminate[5] the lower-level layers: the low-level Beta, or any other language, code is necessary; it is the simplest way to concretely and precisely express the algorithm. However, it is possible to hide this low level behind progressive layers of abstractions which move towards domain-oriented concepts. This way, the end-users without Beta knowledge will understand, and tailor, at the domain-oriented level, while the end-programmers will tailor the lower levels. Furthermore, one can implement domain-oriented environments, say based on visual pro-

[4] one can question if a presentation class is a hypermedia-domain concept or just an implementation concept? I will assume it is a hypermedia concept, for it is not unreasonable to expect a hypermedia user to understand the concept of a presentation for a media.

[5] I a referring to the various attempts to create visual programming languages. Some of them have tried to replace even simple expressions like '2 + 3' by a dataflow diagram which expresses the same thing, only in a lot more space. My personal experience has been that this is really not useful. I believe that while visual languages will work well with higher-level domain-oriented concepts, they will fail miserably if they try to eliminate simple programming-language constructs, like expressions, with visual constructs.

gramming, to support the domain-oriented layers, leaving the domain-independent layers to be programmed using conventional languages.

The study of various language mechanisms has revealed that features such as those present in Beta are well suited for developing tailorable systems. The requirement that extensions refine, and not replace, existing behavior is really a benefit. It leads to extensions that are easier to construct, maintain, and understand. In addition, as has been argued in section 6.6, it leads to extensions that have a better chance at surviving new versions of the system. The ability to write nested patterns also proves useful in implementing systems according to the hardware metaphor. Pattern variables, and patterns as first-class values, have proved central to the approach. They have allowed for the implementation of hardware-like slots and boards. The unification of all abstraction mechanisms into a single pattern mechanism has resulted in a much cleaner approach for developing extensible systems. As a result, the interpreter's API is small but, at the same time, general. It can handle extensions to any aspect of the system.

It is still not clear as to which approach for making a system dynamically tailorable is best. The embedded-interpreter approach has worked well and has the benefit of producing a stand-alone application with low overhead. On the other hand, compiling the extensions is appealing as it would deliver better performance for the extensions. One solution to this might be to develop a low-overhead embeddable compiler with a dynamic linker. In fact, replacing the core of the interpreter with a code-generator would do exactly this.

Introducing dynamic extensibility into a static language has worked well. It has had no influence on the implementation of static programs: they are still implemented as efficiently and safely as they always were. There has been no change in the runtime system of the Beta system. It has not required any significant change in the style of coding of Beta programs. The extensions are also safe: they are checked at load time. The only potential problem is the efficiency of the interpreted code. This work has taken static languages, like Beta, a notch towards dynamic languages like Smalltalk.

In my opinion, the dynamic aspect is essential if we are to build tailorable systems. One could argue that one should just use a language like Smalltalk or Self and not bother with using Beta, especially if it is so much work to add dynamic behavior to it. In some cases, this argument would hold. Beta, however, is claimed to be an industrial language with support for programming-in-the-large. Therefore, if one wanted to build a large industrial system, one would choose Beta over Smalltalk. At the same time, I would like to add that Smalltalk is quickly becoming an industrial language. In addition, recent work has shown how to add strong typing to Smalltalk in an industrial setting. This has moved Smalltalk a notch towards Beta. So, eventually, the choice may depend on the needs of the system one is building, and of course on the availability of trained developers.

Building the underlying domain model for a tailorable system is a complex task. There is a need for extensive domain knowledge, the type of knowledge that average system developers don't have. It is imperative that some of the potential users be involved in this process. During the course of this work, I came to a stage where I had to decide which application domain to test my ideas on. I was interested in building a tailorable system in the financial engineering domain. When I got to evaluating this option, however, I found that my limited domain-specific knowledge was entirely insufficient to make a tailorable system. Even if I could have come up with a limited domain model of the system, it would have been impossible to determine what types of tailorability to support. When I contacted a colleague who had worked in the area of financial systems, he reinforced my feeling that attempting to build a tailorable system in the financial domain, without involving a person who worked in that domain, would not produce anything useful. He suggested that I try to work in some system-development related domain; it was then that I decided to work with the DeVise hypermedia system. The crucial point here is that creating a domain model for a tailorable system is a complex and domain-knowledge-based task.

Building a good tailorable system is definitely an iterative process. Once one has a working system that one can experiment with, new ideas for tailoring it emerge. This was my experience with the hypermedia system and with the many additional experiments I conducted. Once I had used the system, it was much easier to formulate tailorability ideas. The tailorable hypermedia system went through a few revisions before it arrived at the version presented here.

A question I asked my self throughout the work with the hypermedia system, was: how much knowledge of the system does the tailor need in order to tailor it to support a new media type? When I went through the process of understanding the hypermedia system, in order to make it tailorable, I had a number of questions which included the following: What were the primary objects? What was the causal relationship between them? What were the relationships between their lifetimes? What were their responsibilities? Which other objects did they use to support them? How was a media-type represented in the system? Answers[6] to these questions helped me understand the overall structure, organization, and dynamic behavior of the system. All of this information, however, is not necessary for the tailor. In fact, as shown earlier, knowledge of the presentation class and the session manager class are all that is necessary to tailor the hypermedia system. At the same time, this knowledge is definitely not sufficient to gain any real insight into the operation of the hypermedia system. So, should the tailor decide to do some non-standard tailoring, he/she will be inadequately prepared. Therefore, it may be necessary to document more than just the relevant open points.

[6]One of the original developers was available to answer these questions.

The individual papers also have their own conclusions; see Sections 2.7, 4.4, 5.4, and 6.7.

## 1.8 Future Work

### 1.8.1 Learning to develop better tailorable systems

During the course of the dissertation work, a series of experiments were conducted. These involved the drawing editor which was later used for the hypermedia system. The goal of the experiments was two-fold: (1) determine the limits of extension via specialization, and (2) create the requirements for a tailoring environment by recording the types of information used by the tailor in the process of tailoring the drawing editor. The collected data revealed first that there were certain types of extensions that were very intuitive to pose, but were quite difficult to handle. An example was the introduction of a multi-sided polygon to the drawing editor; the editor had built-in support for rectangles, lines, and circles. The problem was that the editor had, built into it, the assumption that each type of graphical object would require a fixed number of coordinates to represent: e.g. a line would need 2, a circle 2, and a rectangle also 2. Therefore, adding a multi-sided polygon, each of whose instances could require a different number of coordinates to represent, while possible in terms of the language, would cause interaction problems with other parts of the drawing editor. Furthermore, there was no simple way to change that built-in assumption without actually redesigning the editor by modifying its source-code. Note that this situation was not setup deliberately; it simply happened as part of the experiment. It is my feeling that there are probably many more such situations in an application of reasonable complexity, and there is need for techniques to either (1) design the system so that such assumptions are not made, or (2) make it possible to modify these assumptions via specializations. Continuing such experimental work with the idea of constructing such techniques would be a useful area of future work.

### 1.8.2 Developing tools for tailoring

Another part of the collected data shows the types of information about the drawing editor used in tailoring it. Attempts were made to tailor the editor in the following ways: add a multi-sided polygon, add a rotate 20 degrees command (the system already had a rotate for other angles), add a new attribute for all graphical objects, add an align command. The data reveals that, in the case of the multi-sided polygon, for example, much information scattered all over the editor's code was used. It was simple enough to see that a polygon should be a specialization of the graphical-object class, and that it

should implement certain methods, but much harder to see how it should interact with its environment — i.e. how it should implement the methods. In fact, not even the original code contained this information. Knowledge of the protocol between graphical objects and their environment was used in order to construct the polygon definition. Hence, there is a need for tools that can record and supply this information in a language-independent manner. There is much to be learnt and understood before such tools can be constructed. An idea would be to take the data collected from the experiment, record the information in a hypertext document, and experiment with this setup.

Part of the information presented by such a tool could be generated automatically from the system's sources, and the rest would have to be provided by the system developer. A number of experiments can be done to determine exactly what can be automatically generated and how. How should the system developer specify this additional information?

### 1.8.3 Implementing new language features

Another interesting area of work is at the language level. We have made it possible to introduce new Beta patterns into an running Beta program, but only at points where there are declared pattern variables. An interesting variation on this would be to allow *any* (not necessarily used through a pattern variable) pattern in a Beta program to be replaced by a dynamically-loaded one. A scheme to handle this could be devised. One would have to resolve problems of the following nature: what should be done to active instances of the old pattern? Should they be upgraded to the replacement pattern, and if so, how? What about objects who have the instances of the replaced pattern as inline objects? How can one upgrade these? The solution to many of these problems lies in a slightly different, more flexible, and less efficient, runtime representation. Can this be avoided? The introduction of virtual patterns as super-patterns is something that was attempted by the original Beta implementations, and abandoned subsequently due to the inefficiencies introduced by it. It is, however, a powerful mechanism in the construction of extensible systems and should be supported. As previous attempts to implement it have revealed that it is expensive to implement, tests should be conducted to determine exactly how expensive it is. It is entirely possible that the benefits will outweigh the cost.

### 1.8.4 Developing Beta systems with metalevel architectures

Metalevel architectures offer the possibility to do a variety of things to the system they represent. These include fine tuning, introspection, and tailoring. How could one introduce mechanisms, to implement systems with metalevel architectures, in a language like

Beta? In particular, an obvious experiment would be to create a general purpose metalevel architecture for the hypermedia system. Just as in the Silica window system [Rao91], the fundamental aspects of the behavior of a window and its relationships with other windows were captured by metaobjects, so should similar aspects of a media-type be captured in metaobjects. This could be a refinement of the metaobjects already present in the hypermedia system. Currently, for a media-type, we have a metaobject which specifies its presentation behavior, its instantiation behavior, and its storage behavior. It is possible to change any of these by specifying a new class for any of them. The behavior of a media-type can, however, be represented at a finer level: e.g. its presentation behavior is composed of its linking behavior, its editing behavior, and its viewing behavior. All of these could be made metaobjects. The advantage would be that all of these would then be manipulable by the tailor. Furthermore, the major hypermedia protocols should also be identified in order to allow the tailor to change relevant functionality.

### 1.8.5 Specifying the specialization interface for open points

Specifying the specialization interface of open points could help in the tailoring process. Taking the hypermedia example again, it should be possible to specify, in some formal or semi-formal manner, the exact interface that a specialization of the presentation class should see. This need not necessarily be the same as the interface that a user of the presentation class would see ([Lam93]). The interface should also include information about the environment visible to the specialization. This is because, in Beta, the open point may be nested in some block, and there may be a number of names visible in its environment. Some of these names may be relevant to the specialization. In the hypermedia, for example, the specialized drawing editor accesses the current session object. Experiments in writing the specification of open points, based on the techniques in [Lam93], would be a useful area of future work.

### 1.8.6 The impact of distribution on extensibility

Assume a system has an object which is an instance of an interpreted extension pattern. Assume, further, that a reference to this instance is made available to another remote system through some distributed object facility. If the corresponding extension pattern has not been installed in the remote system, how should the remote system handle this object? A possible solution may be to use an approach similar to the persistent-store approach presented in Section 6.5. An interesting area of work would be the identification of such problems and the creation of appropriate solutions.

# Chapter 2

# On the Construction of Extensible Systems

**Author:** Jawahar Malhotra

**Date:** August 1993

**Publication Information:** Proceedings of TOOLS Europe 94, Versailles, France, March 1994 [Mal94].

**Abstract.**

Extensible systems are systems which can be extended, by adding new code, without requiring any modification to the original system's source code, without recompilation of the original system, and possibly, without even having access to the original source code. An approach which enables the construction of highly extensible systems is described. It is based on a metaphor: construct the system so it resembles a piece of computer hardware, in that it has a mother-board with slots, and extension boards that plug into these slots. Language mechanisms, that support the construction of systems in this manner, are described. Concrete examples of this approach are presented. The realization of these mechanisms in the programming language Beta, an object-oriented language, are shown. The approach is general enough to be adaptable to other languages. Three alternative techniques for processing extensions are discussed. An attempt is made to classify all changes that can possibly be made to an object-oriented system. This is used to determine which types of changes can be supported, by installing extensions in a possibly redesigned system, rather than by modifying the source code and recompiling.

## 2.1 Introduction

This report describes work in the context of the DeVise project at Aarhus University. The overall goal of the DeVise project is to create concepts and tools for experimental system development. A subgoal is to explore the area of user-tailorable systems. To support tailorable systems, one needs the concept of system extensibility, especially of the dynamic kind. Hence, the area of extensible systems been explored in order to gain some insight into what support a programming language should provide to allow extensible systems to be written.

An extensible system tends to have wider applicability than a rigid one. This is because more people can adapt it to their use situation. It also tends to have a longer lifetime as it can grow with the user's needs. Furthermore, it acts as a catalyst for new ideas: advanced users can easily experiment with various extensions, leading to some very creative uses of the system not conceived of at development time. GNU Emacs [Sta84] is a good example of this.

By an extensible system, is meant one that is executable, yet has the potential of accepting new code; code which may either replace or enhance existing parts. Furthermore, in order to make the extension, there should be no need for the source code of the original system (only symbol-table information is required), nor, preferably, should there be any need for the entire development environment. Needless to say, the source code of the original system must not be modified or require recompilation. In the terminology of [Nør92] who introduces the notion of an *open point* as a behavioral parameter of a program, an extensible system is a system with open points.

When one tries to develop a system under such constraints a number of interesting issues arise.

- **Structuring the System to be Extensible.** A system comprises of a number of components.[1] What should these components be, and how should they be organized, so that significant changes in the behavior of the system can be effected by replacing these components by newer versions? If the components are too fine-grained, simple changes of behavior will involve many components, thus making the extension process laborious and error-prone. If they are not comprehensible in their own right, they will be difficult to extend. A good component-level architecture will also make it easier for the person tailoring the system to gain an overall understanding of the system and to identify which components need to be replaced in order to install a certain extension.

[1] By components is meant the parts (functions, procedures, classes, modules,...) of the application.

- **Writing Open Components.** There is a need to write components in a way such that they may be specializable later without changing their original source code. This is essential if the system is to be extensible without modification of the original sources. Callbacks or hooks are an example of a mechanism for writing functions/procedures which may be extended without modification of the original sources.

- **Ensuring a small Semantic Gap.** End-users typically know what functionality they would like to change or add. It is not very easy for them to find what they need to specialize or to find exactly where they should add it. Hence, there is a need to write the system components in a way such that they correspond, more or less, to the functionality of the system. This will reduce the semantic gap, thus facilitating the location of code. Tool support can also aid this process. So, it is imperative that all the bits and pieces that go together to implement some feature of the system be packaged together, as much as possible, within a component.

- **Installing Extensions.** Once an extension has been written (i.e. an original component has been located, comprehended, and specialized), it is still necessary to make the compiled application use the specialized version instead of the original version. In a language with dynamic scoping (e.g. Smalltalk [GR83]), the replacement could be done by giving the extension the same name as the original. But, in a language with static scoping, this would not work. There is a need for some additional mechanism to support this; e.g. use variables: if the original component was referred to through a variable, then assigning the new extension to that variable would have the desired effect.

It is interesting to examine the types of language mechanisms necessary to handle these issues.

Using an example of a spreadsheet system, and the programming language Beta, techniques and language mechanisms which help write such extensible applications are illustrated. Beta is a strongly-typed, statically-scoped, block-structured, object-oriented language in the spirit of Simula [DMN68].

The hardware metaphor which suggests that a system be structured so it resembles a piece of computer hardware, in that it is has a mother-board with slots and extension boards that plug into these slots, is introduced. This simple metaphor proves surprisingly useful in guiding the overall architecture of the extensible system. Various language mechanisms that support this approach are identified; their realizations in Beta are illustrated and discussed. The author feels that Beta has a number of interesting features that make it particularly elegant for writing extensible systems.

In addition to exploring the construction of extensible systems, this report also explores the extension process; i.e. handling extensions. Extensions must be processed, i.e. compiled, or interpreted, before they can be installed into the extensible system. Three alternatives for processing extensions are explored.

In order to gain an understanding of the power of this approach, an attempt is made to classify all changes that can possibly be made to an object-oriented system. This is used to determine which types of changes can be supported, by installing extensions in a possibly redesigned system, rather than by modifying the source code and recompiling.

Section 2.2 describes, as an example, an extensible spreadsheet system which is characteristic of the types of extensible systems this work tries to support. Section 2.3 presents guidelines for constructing extensible systems and then illustrates how the spreadsheet system can be constructed in accordance with these guidelines, using Beta as the programming language. Section 2.4 covers the alternatives available when it comes to processing extensions. Section 2.5 examines the types of changes one can make to an application and shows if they can be supported by the extensibility mechanisms presented earlier.

## 2.2 The spreadsheet example

As an example, consider an extensible spreadsheet system which allows users to specialize existing cell-types and define new types of cells; e.g. cells, which are designed to contain free-format text, can be specialized to support formatted text, or, a new type of cell which contains video can be defined. These changes or additions should be definable by the user and incorporable into the application executable without recompilation of the original sources.

This incorporation could happen dynamically, during the execution of the spreadsheet system; the spreadsheet system could load the new extension. It could also happen statically; the extension could be patched into the original executable yielding a new extended executable.

It is important to keep in mind that it is the end-user[2] of the spreadsheet system who, without having access to the source code of the system, and preferably without having the original development environment, will write these extensions.

[2] who, probably, should be a programmer, better known as a super-user.

## 2.3 Building the extensible spreadsheet in Beta

The problem then, is to write such an extensible spreadsheet application and to identify what special language mechanisms were used to accomplish this task.

Computer hardware is designed to be highly extensible. This is primarily because it is not very easy and economical to produce completely new hardware each time an extension has to be made. What aspects of hardware make it so extensible? It is primarily the overall architecture which comprises of a mother board with slots. Much of the extensible functionality resides on extension boards which are plugged into these slots. Users can typically handle the following tasks themselves:

- add a new board in an empty slot
- specialize an existing board (e.g. add new memory SIMMs)
- replace an existing board by a newer version

If software could be constructed in this manner, similar benefits could be availed of. One way to look at it is as follows: the mother board of a program is the main program and the slots are open points (e.g. callbacks in many languages). The extension boards are the various software components (classes, procedures, functions, types, ...) which fill the open points. This structure (of mother board, slots and components) can be replicated within each software component.

Extending a software system can then be viewed as a process very similar to extending a piece of computer hardware. With software, in addition, to just swapping boards, it would also be possible to allow users to construct new boards (generally by modifying existing ones). It may also be possible to support modifications of the mother board.

The development of the spreadsheet application in Beta follows this theme: it is constructed in the manner prescribed above. The construction is presented in stages — primarily to illustrate all the features without too much complexity. In the first attempt, is presented a version which doesn't allow dynamic extensibility. It serves to introduce the general framework of the application and acts as a reference with which to compare the dynamically extensible version. The second attempt presents a version which can be dynamically extended.

Appendix A presents a brief introduction to Beta.[3] It is recommended that the reader glance briefly at it.

[3]For a detailed description of Beta see [KMMPN87, MMPN93].

### 2.3.1 The first attempt

Here is an abstract presentation of a spreadsheet system. Only details relevant to this discussion are presented. `SpreadSheetSystem` is declared as a Beta pattern; nested within it are declarations of `Cell`, `TextCell` and `PictureCell`; the latter two are sub-patterns of the first. A few variables, `Row` etc., are declared. Finally, there is the do-part which is the body of `SpreadSheetSystem`; it gets executed when `SpreadSheetSystem` is executed; it creates instances of `TextCell` and `PictureCell`.

**Listing 1.**

```
SpreadSheetSystem:
(#  Cell:                                              (* Cell pattern *)
    (# loc: @Position;
       draw:< (# ...INNER... #);     (* ':<' means virtual declaration *)
       edit:< (# ...INNER... #);
       update:< (# ...INNER... #)
    #)
    TextCell:< Cell                 (* TextCell specialization of Cell *)
    (# value: @Text;
       (* '::<' means further binding of virtual *)
       draw::< (# do ... textOutput value; INNER ... #);
       edit::< (# do ... invoke text editor on cell; INNER ... #);
       update::< (# do ... recalculate value; draw; INNER ... #)
    #);
    PictureCell:< Cell           (* PictureCell specialization of Cell *)
    (# value: @Bitmap;
       draw::< (# do ... bitmapOutput value; INNER... #);
       edit::< (# do ... invoke bitmap editor on cell; INNER ... #);
       update::< (# do ... recalculate value; draw; INNER ... #)
    #);
    Row: [100]^Cell;            (* array of references to Cell objects *)
    Sheet: [100]^Row;            (* array of references to Row objects *)
    aRow: ^Row;                           (* reference to a Row object *)
 do ...
    Sheet[10][] -> aRow[];       (* Get and store REFERENCE to row 10 *)
    (* create a text cell at position (10,10) *)
    new(TextCell) -> aRow[10][];    (* ref to new text cell in col 19 *)
    aRow[10].edit;            (* invoke edit pattern on this new cell *)
 (* create a picture cell at position (10,11) *)
    new(PictureCell) -> aRow[11][];
```

```
    aRow[11].edit;
    aRow[10].update;
    ...
#)
```

To run this spreadsheet system, one must create an instance of `SpreadSheetSystem` and execute it in some runtime environment.

**Listing 2.**

```
main:
(#  SpreadSheetSystem: (# ... #);
    aSpreadSheet: ^SpreadSheetSystem;
 do ...
    new(SpreadSheetSystem) -> aSpreadSheet[];
    aSpreadSheet;                                  (* execute the system *)
    ...
#)
```

If this were a real implementation of a spreadsheet system, rather than just an illustrative one, the do-part of the `SpreadSheetSystem` would invoke some user-interface loop which would present some standard spreadsheet-like interface. Statements such as `new(TextCell)` etc., shown being executed directly in the do-part, would typically be executed indirectly through this user-interface. As the user-interface aspect is not relevant to this discussion, it has been omitted for the sake of simplicity.

When `main` executes `SpreadSheetSystem`, by creating an instance of it and executing that instance, control is in the do-part of the spreadsheet pattern. The `new(TextCell)` creates an instance of the `TextCell` pattern and stores a reference to it in position `(10,10)` of the sheet. When the method `draw`, for example, is invoked on this instance of `TextCell`, execution begins in the `draw` method declared in `Cell`; when it hits the `INNER` construct it enters the `draw` method declared in `TextCell`. Upon completion of this, control returns to the `draw` method in `Cell` at the statement following the `INNER`. The same applies for the other virtual methods in `TextCell` and `PictureCell`.

In this implementation of the spreadsheet system, the following language mechanisms are utilized to implement the system so it is extensible:

- **Block Structure.** Block structure allows one to construct a software system as one would construct a hardware device i.e. as a number of sub-units (boards or components) which are part of a larger unit and so on. As in a piece of hardware,

there are component boards, in a software system there are blocks. Nesting of blocks, as in Beta and Simula, allows one to model the building blocks of a system at a coarser level than just ordinary unnested classes. It also serves to package together related functionality. For example, the `SpreadSheetSystem` pattern models a spreadsheet building block and the `TextCell` pattern models a text-cell building block.

Once a system is structured in the form of such building blocks, making extensions is conceptually easier: replace the appropriate building block by an extended version. This is similar to what happens in the hardware world: a computer's networking capabilities are extended by replacing its existing ethernet board with a newly released version.

- **Virtual Patterns.** A virtual pattern allows one to model a component that is open i.e. has some deferred functionality, yet, possibly has some default behavior. They are essential if one has to write components which must be extensible without any changes to their original definition.

  In Beta, patterns unify types, classes, procedures, and functions. As a result, the concept of virtual pattern in Beta, spans all of these related concepts of virtual types, virtual classes etc. Sandvad and Nørgaard, in their report [NS90], show how virtual patterns can be used to defer actions, values, and substance i.e. declare components in which these aspects may be specified later or their existing definitions specialized, all without modification to their original definitions.

  In the example, `Cell` is written with virtual procedures (deferred action): `draw`, `edit`, and `update`. In addition, `SpreadSheetSystem` is written with virtual classes (deferred substance): `TextCell` and `PictureCell`.

- **Further binding of virtuals in all specializations.** In Beta, unlike Simula and most other object-oriented languages, a further binding of a virtual *does not* override the original definition or previous bindings. For example, in `SpreadSheetSystem` in the declaration of `TextCell`, the further binding of `draw` doesn't override the original declaration of `draw` in `Cell`; it extends it. In a further specialization of `TextCell`, a further binding of `draw` would not override the original declaration in `Cell`, nor would it override the binding in `TextCell`; it would extend them.

  This allows for multiple levels of refinement and is particularly useful as it allows an extension made to an extended component to behave as expected. With overriding, an extension made to an already extended component would end up losing its previous extensions unless it was explicitly programmed to utilize them, thus putting an additional responsibility on the shoulders of the extension programmer.

Put simply, with the Beta model for further bindings, by making an extension it isn't possible to easily break what already works.

- **The INNER construct.** This allows one to specify precisely when, in an extensible action component, control should be transferred to the specialization. It models the fact that the extensible component decides when the extension should be invoked; not the other way around. Extensions don't need to be concerned with specifying when they should be invoked.

- **Nested Specialization.** In other words, the ability to build a new component from an existing one by specializing some nested component. For example, one could specialize the `TextCell` component by creating a new spreadsheet system as a specialization of the old:

  **Listing 3**.

```
NewSpreadSheetSystem: SpreadSheetSystem   (* create new system *)
(#                                  (* identical to old, except: *)
    TextCell::<           (* text cells are slightly different *)
    (#                                  (* in their edit command *)
        edit::< (# do ... issue warning; INNER #);
    #)
#)
```

  A new spreadsheet system has been constructed, reusing the old, and specializing `TextCell` to issue a warning whenever edited. This is equivalent to constructing a new piece of hardware by copying most of the old and replacing some of the old parts by new extended versions. This is an elegant, powerful, and conceptually clear way to specify an extension to a system.

### Building a new executable

In order to build an executable for the `NewSpreadSheetSystem` shown above, it is necessary to write a new `main` (similar to the one for `SpreadSheetSystem`) and to compile and link it into a new executable. For this process, only the object code for the original `SpreadSheetSystem` and the necessary symbol-table information is necessary. No modifications to, or recompilations of, the original source code are necessary.

Another approach that uses Beta's pattern variables (see Appendix A.1), the interpreter (see [Mal93a]), and doesn't require the rewriting, or recompilation, of `main`, can be written as:

Listing 4.

```
main:
(#  SpreadSheetSystem: (# ... #);          (* same as original declaration *)
    theSpreadSheetSystem: ##SpreadSheetSystem;        (* pattern variable *)
  do
    (* load extended version *)
    (extensionFile) -> interpret -> theSpreadSheetSystem##;
    if theSpreadSheetSystem## = NONE then   (* extended version not found *)
        (* install default version *)
        SpreadSheetSystem## -> theSpreadSheetSystem##;
    theSpreadSheetSystem;                (* execute the installed version *)
#)
```

The original `main` has been rewritten to be extensible. An executable obtained by compiling this main behaves as follows: upon startup, it attempts to load an extension from a designated extension file. If found, the extended version of the spreadsheet system is loaded and executed, else the original default version is executed. To extend the system, one simply needs to put the source code for an extension such as `NewSpreadSheetSystem` in the designated extension file and restart the executable. No compilation or linking is required.

The extension is loaded via the interpreter. The return value of the interpreter is a value denoting a pattern. It is like a function-closure; it is called a structure value in Beta terminology. Structure values are stored in pattern variables. In the code above, `theSSSystem` is a pattern variable. These concepts of pattern variables and structure values are discussed in Appendix A.1.

The problems of processing extensions are covered in more detail in Section 2.4.

### Problems with the first attempt

It is possible to construct a new spreadsheet system using the old. It isn't possible to modify the old system in place. To get the effect of an extension, the new system must be started instead of the old. It isn't possible to make extensions dynamically i.e. during the execution of the old system, as it is necessary to abort the execution of the old system and execute the new system.

As a result of this, it is impossible to write a spreadsheet system which has a command, which allows its user to specify the code for a new cell type and, upon completion, allows the system to resume execution with the new cell type defined.

### 2.3.2 The second attempt

To make the spreadsheet dynamically extensible, we must define it as follows. Lines marked with a vertical bar are changed/added with respect to the first attempt.

**Listing 5.**

```
DynExSpreadSheetSystem:
(#  Cell:                                              (* Cell pattern *)
    (#  loc: @Position;
        draw:< (# ... #);
        edit:< (# ... #);
        update:< (# ... #)
    #)
|   TextCellDefault: Cell
    (#
       >>Identical to definition of TextCell in SpreadSheetSystem<<
    #)
|   PictureCellDefault: Cell
    (#
      >>Identical to definition of TextCell in SpreadSheetSystem<<
    #)
|    TextCell: ##TextCellDefault;                    (* pattern variables *)
|    PictureCell: ##PictureCellDefault;
|    xxxCell: ##Cell;
     Row: [100]^Cell;
     Sheet: [100]^Row;
     aRow: ^Row;
 do  ...
     (* store defaults in pattern variables *)
|    TextCellDefault## -> TextCell##;
|    PictureCellDefault## -> PictureCell##;
     ...
     (* create a new text cell via pattern variable *)
     new(TextCell) -> aRow[10][];
     ...
     (* create a new picture cell via pattern variable *)
     new(PictureCell) -> aRow[11][];
     ...
     (* create a new xxxCell *)
|    new(xxxCell) -> aRow[12]^;
```

```
    ...
#);
```

`TextCell` and `PictureCell` have been renamed to `TextCellDefault` and `PictureCell-Default`. Pattern variables, with names `TextCell` and `PictureCell`, and types `TextCell-Default` and `PictureCellDefault` respectively, have been declared. The default patterns for text and picture cells (i.e. `TextCellDefault` and `PictureCellDefault`) have been stored in the corresponding pattern variables. With the exception of the use of the pattern variable `xxxCell`, this version of the spreadsheet (`DynExSpreadSheetSystem`) will behave identical to the previous version (`SpreadSheetSystem`).

`DynExSpreadSheetSystem`, however, leaves open the possibility of change in the definition of its text and picture cells, as well as the introduction of a new cell type, all without any modification, or recompilation of its source code. If the following statement were compiled into the above application, executing it would change the pattern denoted by the pattern variable `TextCell` to a dynamically-loaded extension (which must be sub-pattern of `TextCellDefault`).

**Listing 6**.

```
(TextCell Extension Source Code) -> interpret -> TextCell##
```

Here, `TextCell Extension Source Code` is the source code of this extension, presumably obtained from the user. Execution can resume after this statement; there is no need to restart the entire spreadsheet system. All uses of `TextCell` will see the new extended version once this statement has been executed. An example of a text cell extension could be the following:

**Listing 7**.

```
TextCellExtension: TextCellDefault
(#  edit::< (# do ... issue warning; INNER; #)
#)
```

If in an execution of `DynExSpreadSheetSystem`, the above extension were loaded, all subsequently created text cells would issue a warning every time they were edited.

Using pattern variables and the interpreter in this manner, it is possible to write systems which allow parts to be dynamically replaced by specialized parts, all without any modifications, or recompilations of the original source code. It isn't even necessary to restart the application.

This example also defines the pattern variable `xxxCell`. This is defined in order to allow the introduction of new types of cells, as opposed to changing the behavior of existing types of cells. If the following statement were compiled into `DynExSpreadSheetSystem`, executing it would store a new dynamically loaded pattern (which must be a sub-pattern of `Cell`) in `xxxCell`.

**Listing 8.**

```
(NewCell Source Code) -> interpret -> xxxCell##;
```

Once again, `NewCell Source Code` is the source code of the new cell type, obtained from the user, and could contain, for example, the following:

**Listing 9.**

```
VideoCell: Cell
(# value: @Video;
   draw::< (# ... playVideo value; INNER ... #);
   edit::< (# ... invoke video editor on cell; INNER ... #)
   update::< (# ... recalculate value; draw; INNER ... #)
#);
```

`DynExSpreadSheetSystem` could load such a definition, enabling it to support video cells, a feature that was not conceived of at development time.

Pattern variables truly allow one to model the concept of slots in hardware. Given a system, we can replace what is contained in a pattern variable or add something into an empty pattern variable, all without building a new system. This is exactly how it is in the hardware world.

Virtual patterns don't accomplish this. A virtual pattern also denotes a kind of slot, but a slot with something already built into it; that pre-built part cannot be removed or replaced. To put something new into this slot, one must replicate the entire machine; the new component then becomes a pre-built part of the same slot in the replicated machine. This is effective in software as this replication process is relatively easy and cheap.

### 2.3.3 Comparison with other languages

It is interesting to see how extensions can be defined in procedural languages such as Pascal [JWMM85] and C [KR88]. In these languages, everything is modeled as types

and procedures (or functions). For example, one may model a Cell by declaring a record type, a create function which returns newly allocated space for a Cell record, and functions/procedures which model the operations that can be performed on Cells. We can define an extension of a Cell (e.g. to a VideoCell) by defining a new record type called VideoCell which must include all the fields of a Cell along with any additional fields.

In other words, it isn't possible to construct software in the "hardware style" advocated earlier. It is possible to view procedures/functions as the component boards and procedure/function variables as slots; this is not very useful for extensibility purposes as the granularity of the components is very fine. Furthermore, other features useful in the construction of extensible systems, e.g. reuse, small semantic gap, etc. are not easily realizable in these imperative languages.

Other OO languages, such as Smalltalk [GR83], C++ [ES90], and CLOS [Kee89], support reuse, easy definition of extensions, small semantic gap, etc. Smalltalk, and CLOS, due to their dynamic nature are very suitable for writing, processing, and installing dynamic extensions. With CLOS, in addition, it is also possible to use the Meta Object Protocol [KdRB91], a powerful means to extend and tailor a system.

All of these languages, however, lack some of the features identified as useful for constructing extensible systems. These include block structure, virtual patterns (as opposed to just virtual methods), further bindings, the inner construct, and pattern variables.

### 2.3.4 Summary

An approach for constructing statically and dynamically extensible systems has been presented. The approach is based on constructing software in a manner analogous to hardware in order to attain high degrees of extensibility. The primary language mechanisms that make this type of construction possible are identified, related to the hardware metaphor, and discussed.

Using this approach, and the language Beta, extensible systems can be elegantly constructed. Furthermore, the construction of extensions is also simple and elegant.

## 2.4 Processing Extensions

Given the executable for an extensible system (along with some signature information) and the source code for an extension, it must be possible to process (compile or interpret) the extension in the context of the extensible system. The goal is to arrive at a new executable which includes the extension.

If the extension is to be processed statically (Section 2.3.1), then a compiler, which can handle separately-compiled parts, and a linker are all that is needed. The extension can be compiled as a separate module using the signature information about the object code that is part of the executable; it can then be linked with the original executable to produce the new executable.

Processing the extension dynamically is a little more difficult. It should be possible for an executing application to load the extension into itself and continue execution. A number of alternatives are possible; here are a few:

1. Ordinary (non-incremental) compiler with a dynamic linker and loader.
2. Incremental Compilation and Execution environment.
3. An interpreter compiled into the extensible application.

### 2.4.1 Ordinary compiler with a dynamic linker and loader

When the application needs to load an extension it invokes the compiler, providing it the source code of the extension. The compiler may or may not be embedded within the application. If it isn't, then it can be invoked via the appropriate inter-process-communication routines. The compiler must be capable of handling separately compiled units; it is the responsibility of the application to package the extension as a separately-compilable unit. Once the compilation is complete, the application invokes a dynamic linker and loader to load the resulting object code into the current execution. The loader provides the application with a reference to the extension. This is used to incorporate the extension into the application's execution.

This approach was adopted by [AF89] in their attempts to introduce extensibility into Beta applications. In their implementation, an extension is packaged as a separately-compilable module upon which a compile process is started. The compiled code is then linked and loaded into the – possibly executing – system.

### 2.4.2 Incremental Compilation and Execution environment

Given a set of modifications to an application's source code, and its executable, an incremental compiler is capable of producing a new executable with as little recompilation as necessary. Furthermore, some environments (such as the Mjølner Orm system [Mag93]) are even capable of resuming the original execution after such a recompilation.

In a system like the Mjølner Orm System, the extensible application would have to be executed within the provided execution environment. An extension could be installed by using the environment to suspend execution, incrementally compile the extension and produce a new executable, and continue execution. Note that in this scenario it is the environment and not the application that is responsible for installing the extensions.

This is clearly a useful approach; it is also more general, as arbitrary changes to the original application can be supported. The types of extensions presented in this report don't require any changes to the source code of the original application; they don't require the generality of incremental compilation. Hence, if one is only interested in these types of extensions, incremental compilation is a bit of an overkill.

### 2.4.3 Interpreter compiled into the extensible application

The interpreter is embedded within the application. When the application needs to load an extension it invokes this interpreter, providing the source code of the extension as an argument. The interpreter processes this source code, creates appropriate runtime objects in the current execution's image, and returns a reference to this interpreted entity. The application then refers to the extension via this reference.

If the extension refers to a symbol already defined in the system being extended, the interpreter should be able to resolve this to the appropriate compiled part of the system. It should not require the referred part to also be interpreted.

Should the extension refer to a symbol in a library not already part of the system being extended, the interpreter should be able to incrementally load and link the necessary libraries into the extensible system before resolving the name to the appropriate compiled part.

It is also possible to design the interpreter to exist as a separate entity i.e. not embedded within the application. The application could then interact with it as it does with the compiler and dynamic linker/loader.

#### The Beta interpreter

An embeddable interpreter for Beta has been built and is described in [Mal93a]. It is written almost entirely in Beta, has an object-oriented design, and interprets Beta Abstract Syntax Trees directly. Compiled code may invoke interpreted code and vice-versa. The interpreter is packaged as a library with an Application Programmer's Interface (API) which includes the following:

**AddDecl:** `(context * declText) -> ()`
**MakeDeclExecutable:** `(context * originObject * declText -> executablePattern)`

These are functions which a Beta program can use to load extensions via the interpreter. The function `MakeDeclExecutable` returns a reference to the interpreted pattern; a reference which can be executed. To build a Beta application which uses the interpreter, one simply includes a file defining the interface to the interpreter, and uses the above functions. When the Beta system links the application, the interpreter is embedded within it.

## 2.5 Extensions as viewed from the Inheritance Hierarchy

It is useful to classify the different types of changes one can make to an application. One can then determine if it is possible to program the application so that these classes of changes can be accommodated without modification to the original sources i.e. by simply loading extensions into the application. In other words, it helps answer the question: what kind of changes, to the application, can be made by simply loading new extensions?

For an object-oriented application, one possible way to classify changes is to view them as different types of modifications of the application's inheritance hierarchy. The following list contains such a classification of changes along with suggestions for how they could be accommodated in Beta.

1. Modification of a leaf node i.e. a pattern not used as a super-pattern. If the modified leaf node *cannot* be expressed as a sub-pattern of the original, then it becomes necessary to replace the original by the modified at the source-code level and recompile.

   If the modified leaf node *can* be expressed as a sub-pattern of the original, then this can be supported. The modified leaf node can be loaded as an extension and all points in the executable that referred to the original can be made to point to the new e.g. by using a pattern variable. For example, if one wanted to handle modifications of a pattern `P`, which was not used as a super-pattern, by loading extensions, one could declare a pattern variable `PV` of type `P` and replace all uses of `P` with `PV`. Then, modified versions of `P` could be loaded and, provided they were sub-patterns of `P`, could be assigned to `PV`. The resulting application would behave as if it had been compiled with the modified `P` instead of the original.

2. Modification of a non-leaf node i.e. a pattern used as a super-pattern. Once again, if the modified version *cannot* be expressed as a sub-pattern of the original, then the change has to be made at the source-code level and the application recompiled.

   If it *can* be expressed as a sub-pattern, it is possible to load it as an extension and, with one exception, update all references to the original by the new version. The exception: patterns for which the original was a super-pattern cannot be updated to have the new version as a super-pattern. This is because, in Beta, super-patterns cannot be dynamically changed. This is a limitation with the implementation of Beta.

   It seems logical to use pattern variables to solve this problem in the following way:

   **Listing 10**.

```
WindowP : Component(# #)
Window : ##WindowP;                                  (* pattern variable *)
Dialog : Window (# ... #);                     (* pattern variables used *)
TextEditor : Window (# ... #);                      (* as super-patterns *)
```

   With this setup, one can dynamically change the super-pattern of `Dialog` and `TextEditor` to a specialization of `WindowP` by assigning to the pattern variable `Window`. A modification of `WindowP` could be loaded as an extension and stored in `Window`; as a result all uses of the pattern variable `Window` would get the updated version. The problem is that the current implementation of Beta doesn't allow pattern variables to be used as super-patterns. It is possible to implement this feature; it would add overhead which would be experienced even when it was not used, a good reason to disallow it.

3. Addition of a leaf node. This amounts to loading a new pattern, which could be a sub-pattern of some existing pattern, and executing it. This can be handled easily.

4. Addition of a non-leaf node. This amounts to loading a new pattern, which is a sub-pattern of some existing pattern, and making it the super-pattern of some existing pattern. If it were possible to dynamically change super-patterns, this type of change could be accommodated.

5. Other modifications of tree structure e.g. change the super-pattern of a pattern. These cannot be supported without making source changes and recompiling. Once again, dynamically changeable super-patterns would solve this problem.

## 2.6 Related Work

[NS90] present a detailed discussion of reusability and extensibility in the Beta system. Their work has also been a source of inspiration for this work. Their focus is "extensibility of software components with the point of view of reusability." In other words, if one has a library with a class defining a general purpose window, how can one reuse (in another application) the class while extending and customizing it to one's needs. Although we also need such capabilities, we are also interested in how these extensions can be made in a "ready to run" or running application. This work has built upon their work, especially in the area of using virtual patterns to defer specification of action, substance, and values.

Another inspiration for this work has been the report [Nør92] which explores the concept of an *open point*. An open point is defined as a behavioral parameter of a program. With respect to Beta, Nørmark doesn't talk about pattern variables and also states that "...open points simulated via virtual patterns are closed too early relative to the execution of the program." This report shows that with the availability of pattern variables and the Beta interpreter, this is no longer true. Nørmark also discusses the issue of documenting open points.

Meyer, in his book [Mey88], addresses extendibility as an external quality factor of a software system. He describes extendibility as "the ease with which software products may be adapted to changes of specifications." He identifies the concepts of polymorphism and dynamic binding as concepts that support extendibility. The two principles proposed by him as essential for improving extendibility, *design simplicity* and *decentralization*, are fully supported by the views presented in this report.

The seminal system and paper on extensible systems is [Sta84]. Stallman presents a list of language requirements for the purpose of writing extensible systems. Among these, he lists dynamic scoping and no typing as essential. This work shows that this is not entirely true; it demonstrates that extensible systems can be written in a static-scoped language with static type-checking.

Notkin and Griswold [NG87] describe an extension interpreter which relies on call arbitration, dynamic linking, and multi-language extensions. The extension interpreter is able to load an application's object code, given its interface specification. It is also able to interpret one or more extensions, written in any extension programming language. The extension(s) may use the procedures of one or more loaded applications, thus extending each of them and creating a single extended application.

In [OH92], the authors show how the inheritance hierarchy of an application can be extended by merging it with another, possibly sparse, extension hierarchy. An interesting

discussion of extensible visual formalisms, including extensible spreadsheets, is presented in [JNZM93, ZC92].

On the comprehension of systems, in order to tailor them, Mørch [Mør93] argues that the rationale for the system design should be coupled with the system and provided to the person tailoring the system. Ossher [Oss87] also discusses a mechanism for specifying the structure of large systems; this could be prove useful for gaining an understanding of the system before attempting to tailor it.

Reflective techniques are also useful for writing extensible systems. The more an implementation is reflected, the more information there is available at runtime, thus making it easier to process and install extensions efficiently. The book [KdRB91] presents an excellent and detailed discussion of the use of reflective techniques for making extensions in CLOS. There is also some work on reflection and C++ [CM93]; here a version of C++ called OpenC++ is presented. In OpenC++ classes, and selected methods within them, may be declared reflective. Reflective classes have metaobjects which can be used to extend or change the semantics of method calls.

The idea of pattern variables was originally introduced in [AF89]. In their report, Agesen and Frølund present a mechanism for building extensible systems using dynamic linking and loading. We have built upon their ideas in coming up with a general extensibility mechanism for Beta.

In practical use today are applications on the Apple Macintosh which utilize the concept of "add-on." So, an application extends itself dynamically by loading an add-on which blends in with the original application. The add-on shares the original system's runtime environment. As an example, spelling checkers are generally add-ons which are loaded by word processors when needed. The word-processor and the spell add-on communicate using some event mechanism. The Macintosh add-on concept can be elegantly modeled using the mechanism we provide.

**Listing 11**.

```
WordProcessor:
(#  SpellCheck: ##AddOn;
 do ...loadAddOn -> SpellCheck;
    ...SpellCheck...
#)
```

Here `AddOn` is a predefined pattern which describes the interface common to all AddOns; `loadAddOn` loads a precompiled version of `SpellCheck`; we could also have used the interpreter instead. Then, a spelling checker can be defined and loaded dynamically into the word processor.

[PSS87] presents a discussion of the Beta shadow language. Their focus is primarily in designing techniques to monitor and debug programs. In a sense, they want to support the age-old debugging technique of putting print statements into programs without actually modifying the source code of the program or recompiling the program. This is equivalent to extending the original application without changing its source code or recompiling it, something similar to what has been described in this report. They propose a runtime representation amenable to extensibility e.g. consult a jump table after each statement. Although interesting, such an approach would exact a high efficiency overhead. [BO86] also presents techniques useful for monitoring and debugging of, primarily concurrent, Beta programs.

[TMH87] describe the concept of system adaptability. According to them, a system can be adaptable in four ways: it can be (1) flexible (2) parameterized (3) integratable (4) tailorable. Our focus has been primarily in finding techniques to support (3) and (4). Although we don't say much about these concepts in this paper, we agree that these are useful concepts in the world of the application builders and users. We feel that the choice of how the application should be adaptable and what tailorability language should be presented to the end-user is left entirely up to the application designers and builders.

## 2.7 Conclusions

This report has identified techniques and language mechanisms useful for constructing extensible systems. The usefulness of these mechanisms is premised upon the belief that computer hardware is highly extensible in its construction and hence, software that can be constructed in a similar manner will also be highly extensible. This simple metaphor has proven surprisingly useful in guiding the overall architecture of the extensible system.

This approach to writing extensible systems relies heavily on the power of object-oriented programming in creating highly reusable, generic, polymorphic, and encapsulated components. In particular, it is the polymorphic nature of the extensible application's code (its pattern variables) that allows one to replace a pattern by its sub-pattern, in the executable, without requiring a recompilation of the code that uses the pattern.

The usefulness of these mechanisms has been backed by examples written in Beta. Elegant techniques for writing extensible systems and for extending them have been demonstrated. In addition to the illustrative examples presented here, these concepts have been tested in much larger practical examples. In all cases, they have worked as expected. The approach and mechanisms are not limited to Beta; they can be adapted to other languages.

For processing extensions, the interpreter approach has been successful. The interpreter

has also proven useful for a number of additional applications, like an incremental execution environment, and a source level debugger. The overheads of embedding the interpreter into an application have not been much; for most reasonable sized applications, it has not even doubled the size of the executable.

Extensibility is generally introduced at the expense of efficiency. It may be acceptable for the extensions to be less efficient than pre-built parts. It is, however, less desirable to sacrifice the efficiency of pre-built parts in order to allow them to be extensible. A system which doesn't use the extensibility features shouldn't have to pay for them, while one that does use them should still run acceptably efficiently.

In the presented approach, open points are introduced by using pattern variables. These introduce a level of indirection and, hence, some overhead which is experienced even when pre-built parts are used. A solution would be a scheme which doesn't require all uses of a pattern to be via a pattern variable, yet allows the pattern to be dynamically replaced. This could be done if one could know all the use points, in the executable, of the pattern. One could then patch these to correspond to the new pattern.

## 2.8 Future Work

Object-oriented modeling techniques are geared for producing highly reusable components. These components are designed to be reusable as library components — they can be reused when constructing a new application, or another part of the same application. This is useful in designing components for extensible systems; here also, components are reused, albeit in a different manner. One can view the process of extending a system as identifying a component, reusing it to make a new specialized version, and replacing the original with the new version. It is, however, possible that certain bits of functionality of the extensible application, that would normally *not* be modeled by the standard techniques, need to be modeled when designing an extensible system. For example, in a spreadsheet system, using standard techniques, one may blend the constraint solver into the functionality of the cells. But this would not be so desirable if one wanted to make this constraint solver extensible. We plan to study how the standard techniques need to be modified to produce models for extensible systems.

Tailorability issues come heavily into play while mapping this object-oriented model into an implementation. It is how the various modeled components are put together into a working application that determines how easily one will be able to replace them with specialized versions. We plan to explore techniques for mapping object-oriented models into implementations that are tailorable.

As an end-user of an application, one is not so aware of the internal program structure of the application. One deals with concepts in the application domain and not in the programming domain e.g. it is easier to phrase a statement like "when I select that window" rather than "when that window object receives a mouseDown message followed by a mouseReleased message." Object-oriented programming has reduced this gap between application-world concepts and programming-world concepts by allowing real world objects to be modeled by program objects. There still remains, however, quite a gap between these two worlds. We would like to build tools which allow users to identify extension points in terms of the functionality-view of the application rather than the source-code view of the application. We are also interested in developing techniques and tools which enable the user to comprehend the system being tailored.

The concepts presented in this report are being tested currently in the context of a large application: the DeVise Hypermedia System. This application has already been built in Beta and doesn't have any explicit support for being extensible. It is now being made extensible using the approach presented here. It is hoped that this experiment, in addition to yielding a highly extensible hypermedia system, will also lead to some design guidelines for extensible systems and some ideas for languages and environments for writing extensions.

The relation between reflective techniques and extensibility is also an item of future work. It is clear that much can be gained, in terms of extensibility, by reflecting the implementation of a system. The interesting issues will be to determine how this can be done efficiently.

### Acknowledgements

This work has been generously supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. This work has benefited from the contributions of Prof. Ole Lehrmann Madsen, Görel Hedin, Kaj Grønbæk, Randall Trigg, Anders Mørch, Jørgen Lindskov Knudsen, Elmer Sandvad, Jørgen Nørgaard, as well as many members of the DeVise project and the employees of Mjølner Informatics.

# Chapter 3

# Dynamic Extensibility in a Statically-compiled Object-oriented Language

**Author:** Jawahar Malhotra

**Date:** July 1993

**Publication Information:** Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93), Kanazawa, Japan, November 1993 [Mal93a].

**Abstract.**

Statically-typed object-oriented compiled languages, like Simula, Beta, Eiffel, are desirable because of the safety and efficiency of the resulting code. Dynamically-typed, interpreted languages, like Smalltalk, are useful as they provide the possibility of dynamically extending a program. In this paper, we reconcile the safety and efficiency goals of compiled languages with the benefits of interpreted languages by presenting an embeddable interpreter for a compiled language, namely Beta. The interpreter is designed to be embedded into any compiled Beta application, thus enabling it to accept dynamic extensions. This paper examines the Application Programmer's Interface to the interpreter and illustrates some aspects of our implementation.

## 3.1 Introduction

Statically-typed object-oriented compiled languages — e.g. Simula [BDMN73], Beta [MMPN93], Eiffel [Mey88] — are desirable because of the safety, efficiency, and readability of the resulting code. Dynamically-typed, interpreted languages — e.g. Smalltalk [GR83] — are useful as they provide, among other things, the possibility of dynamically extending a program. In other words, it's possible to incorporate new code into an already compiled, and possibly executing, application without recompiling the application.

Our goal is to allow type-safe dynamic extensibility in a compiled language, namely Beta, without sacrificing the efficiency of the already compiled parts. We want to be able to write Beta applications that can extend themselves dynamically. One technique for supporting dynamic extensibility is to build an incremental compiler as has been done for Simula in the Mjølner Orm system [Mag93]. Another is to build an interpreter which can interface with the compiled parts of the application. We have taken the interpreter approach for Beta; we have built an interpreter which can be embedded into a compiled Beta application i.e. it can be invoked from a compiled Beta application to dynamically interpret programs in the context of the application. The interpreter is provided as a library with an Application Programmer's Interface (API) and must be linked into the compiled application.

Our motivation for introducing dynamic extensibility in Beta is our need for building dynamically tailorable applications. We envisage, developing in Beta, a class of applications similar in terms of customizability and extensibility to GNU Emacs [Sta84]. It is an established fact that object-oriented techniques are well suited for writing extensible software: classes may be extended by defining subclasses. Our goal is to exploit these techniques in a dynamic setting.

As an example of such an application, consider an extensible spreadsheet application which allows end-users to define new types of cells. The application has a command named Define New Cell. The user defines a new cell type by executing this command and providing the Beta code for the class declaration of the new cell type e.g. a VideoCell. Upon completion of this command, the application extends its Create command to include VideoCell as one of the possibilities. Selecting VideoCell creates a video cell (or converts an existing cell into a video cell) using the user-supplied declaration.

Given such functionality, users will be able to customize the spreadsheet application according to their own needs, or extend it with functionality they find lacking. The use of object-oriented techniques, such as specialization and virtuals, will allow them to reuse parts of the original application, thus minimizing the effort required to write the extensions. While such extensions may be installed by recompiling the entire application

with the extensions, the ability to install the extensions dynamically is far more appealing to end-users.

The interpreter is a means of processing these extensions dynamically. The spreadsheet application is compiled with the interpreter embedded within it. The Define New Cell command uses the interpreter to process the user-supplied Beta code.

We use the word *extension* rather than *modification*. An interpreter supporting dynamic modifications would allow for the possibility of dynamically replacing compiled code with interpreted code, thus allowing a self-modifying program to be written. This is not our goal, although it would be a useful side-effect for a certain class of applications such as development environments and debuggers. We touch upon this issue again later.

This work shows that it is possible to write a dynamically extensible application in a compiled, statically-typed, block-structured, object-oriented language; an application which allows the extension language to be the same as that in which the original application was written. This is accomplished without sacrificing any of the good properties of the language e.g. typing and block-structure. Furthermore, it shows that this can be accomplished, without much overhead, by embedding an interpreter into the extensible application. It presents a small, yet general, API for the interpreter. To the best of our knowledge, such an embeddable interpreter is not available for any of the other well known compiled object-oriented languages.

### 3.1.1 Background

The interpreter is part of the Mjølner Beta System (MBS) [MI92b]. All programs in the MBS are stored in Abstract Syntax Tree form (AST).[1] The static-semantics checker works on ASTs too; it is responsible for type-checking as well as generating a symbol-table. In the MBS, this symbol-table is not a separate entity; it is encoded directly into the AST. The code generator emits assembly code in the appropriate assembly language e.g. MC68000 assembly on the HP Series 400 machines. To produce an executable, the MBS relies on tools from the environment in which it is running. For example, under UNIX, it uses the standard assembler, linker, and loader to assemble, link and execute programs. Both compiled Beta modules and executable Beta programs conform to the `a.out` file format (under UNIX).

[1] The reader may safely think of ASTs as source code text except in places where we talk of semantically decorated ASTs. In that case it is equivalent to the source code plus the symbol table.

### 3.1.2 Overview

Dynamic Extensibility using an embeddable interpreter is illustrated further in Section 3.2. The interpreter API, and the implementation of the supporting interpreter core, are presented in Section 3.3. The example presented here illustrates the use of the interpreter in a dynamically extensible system. The interpreter may also be used to build an interactive development environment or to enhance the functionality of a debugger. These applications are shown in Section 3.4. The applicability of these concepts to other languages is also discussed (Section 3.5).

In order to be comfortable with the examples presented, it is recommended that the reader glance at the brief Beta primer (Appendix A).

## 3.2 Achieving Dynamic Extensibility — the general idea

Figure 3.1 presents an overview of the interpreter and its environment in the context of the spreadsheet example. This view is abstract; it is incomplete with respect to many details. Its purpose is to present an idea of how the interpreter fits into the traditional model of an application. It does this by showing the relevant parts and their interactions (invocations and information access). The interpreter is shown embedded within the Spreadsheet application (*host-application*). The large box labeled SPREADSHEET APPLICATION is the executable produced by the Beta compiler from the AST labeled SPREADSHEET APPL. (this is the source code for the spreadsheet application). The interpreter comprises of the Application Programmer's Interface (API) via which the host-application invokes the interpreter to install new definitions, and the core which is the part responsible for actual interpretation. The interpreter is designed to work directly on semantically decorated Abstract Syntax Trees (ASTs); no intermediate code is generated. The ASTs labeled VideoCell and AudioCell are user-supplied dynamic extensions to the spreadsheet application; they contain declarations for the corresponding classes.

Within the application, the Define New Cell command implementation invokes the interpreter, via its API, providing it with the code to be interpreted and some context information i.e. the context (within the host-application's source code) in which interpretation should take place (arrow 1; Figure 3.1). Let's assume this code is the AST VideoCell. The interpreter pre-processes the AST VideoCell, producing some run-time "entities" which allow compiled parts of the application to view VideoCell as if it were also compiled. The box labeled VideoCell represents such an "entity" produced by the

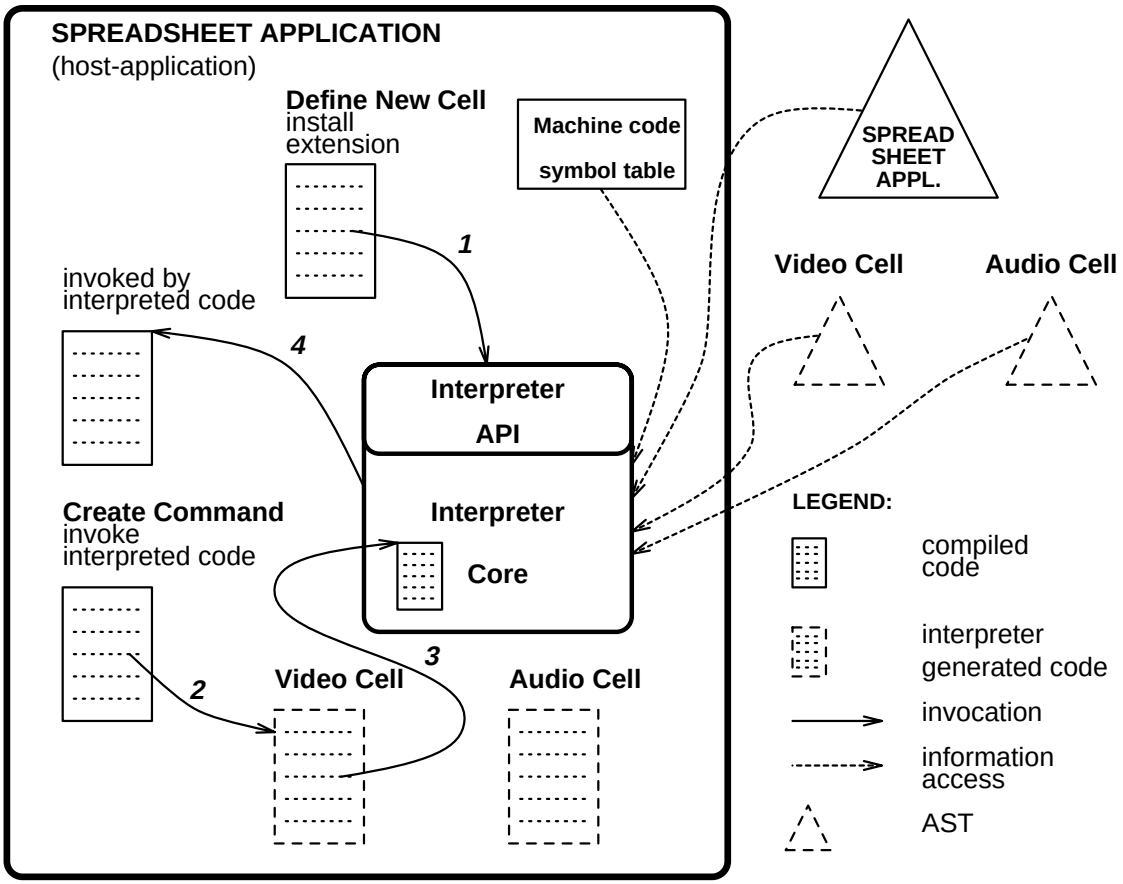


Figure 3.1: The interpreter and its environment

interpreter. With this, we can say that the application has been extended by VideoCell. A reference to this run-time "entity" is returned to the caller (arrow 1, backwards). One may think of this return value as a closure denoting the class (similar to function-closures in functional languages). We also assume that Define New Cell extends the Create command with an option for video cells which uses the returned reference to create them.

When the user attempts to create a video cell, the implementation of the create command invokes VideoCell via the returned reference (arrow 2). Control is transferred by VideoCell to the interpreter core (arrow 3) which accesses the AST for VideoCell and interprets it. During interpretation, it may come across references to compiled parts of the host-application. It resolves these into addresses by accessing the host-application AST and the machine-code symbol table. It then accesses the compiled code (arrow 4).

## 3.3 The Interpreter API

The Application Programmer's Interface (API) to the interpreter defines the necessary functions (known as patterns in Beta) for applications to interface with the interpreter.

It has been designed to allow the following operations on the host-application:

- *Add* new pattern (class) definitions to any context (block).
- *Execute* a new pattern (class) in any context (block).

Intuitively, a context denotes a block in the source program; it is a static property of the program; it is *not* an object. The API comprises of the following patterns (functions):[2]

```
AddDecl              : (context * declText) -> ()
MakeDeclExecutable   : (context * originObject * declText)
                                        -> executablePattern
getCurrentContext    : () -> context
getEnclContext       : context -> context
getCurrentObject     : () -> object
getOrigin            : object -> object
```

`AddDecl` can be used by an application to add a new pattern declaration to itself; this declaration then becomes available to subsequent declarations within its scope; it is not directly executable by the caller of `AddDecl`. `MakeDeclExecutable`, in addition to adding the pattern declaration (like `AddDecl`), also returns, to its caller, a reference to the pattern; this reference may be used to execute (instantiate/invoke) the pattern. The other functions in the API are support functions used to compute context and object arguments for `AddDecl` and `MakeDeclExecutable`.

Returning to our spreadsheet example, one may define VideoCell and AudioCell by first defining an abstract super-pattern called MultiMediaCell as a specialization of Cell (assume that `Cell` is declared by the spreadsheet application), and then defining VideoCell and AudioCell as specializations of MultiMediaCell. In this case, `AddDecl` should be used to process MultiMediaCell while `MakeDeclExecutable` should be used to process VideoCell and AudioCell. The reason: the spreadsheet application never needs to execute MultiMediaCell directly, while it does need to execute VideoCell and AudioCell. We will assume that the spreadsheet application has a command called Define Abstract Pattern which may be used to define an abstract super-pattern like MultiMediaCell. This is in addition to the Define New Cell command discussed earlier.

[2]The signatures presented here are informal and are only intended to convey the functionality of the pattern. They are *not* in Beta syntax.

### 3.3.1 AddDecl

This function is used by an application to add one or more new pattern declarations to a given context. The `declText` argument is the text of the new pattern declaration. The function `getCurrentContext` should be used to obtain a context. The following example and the accompanying explanation illustrate the idea. Here `Spreadsheet` is the pattern (class) describing the spreadsheet application.

```
Spreadsheet :
(#  Cell : (# ... #);              (* Cell is an abstract superclass *)
    TextCell : Cell (# ... #);     (* TextCell is a subclass of Cell *)
    (* loads an abstract pattern declaration *)
    DefineAbstractPattern :
    (#  c : @context;
        decl : @text;
     DO getCurrentContext -> getEnclContext -> c;
        loadPatternDecl -> decl;
        (c, decl) -> AddDecl;           (* declare decl in context c *)
    #)  (* DefineAbstractPattern *)
#)  (* Spreadsheet *)
```

The functionality of the Define Abstract Pattern command is captured by the pattern `DefineAbstractPattern` shown above. As an example, the following declaration could be loaded using Define Abstract Pattern.

```
MultiMediaCell : Cell (# ... #)
```

The effect of calling `DefineAbstractPattern` in this scenario is shown in Figure 3.2; here the extended version of the spreadsheet application's source code is shown. Another way to think about this is as follows: the spreadsheet application, after dynamically adding `MultiMediaCell`, is equivalent to an application obtained by compiling the contents of Figure 3.2.

#### AddDecl details

The execution of `DefineAbstractPattern` proceeds as follows:

1. `getCurrentContext` yields the current context. In a statically-scoped, block-structured language like Beta, the context in which a declaration is processed influences the outcome. A context is static information about the program itself. Intuitively, it denotes a block in the source program. In our case, as all Beta programs

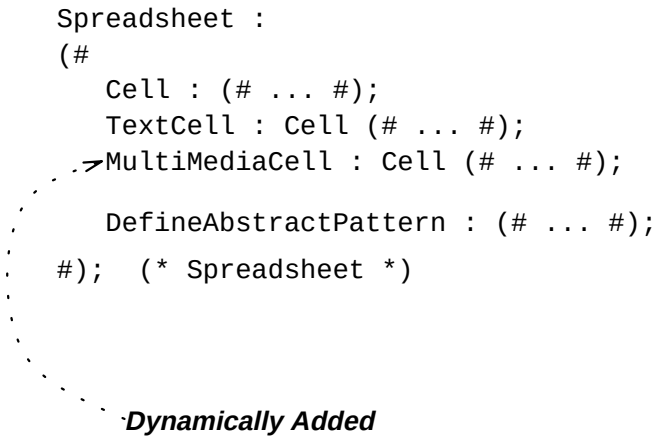


Figure 3.2: Illustrating the effect of `AddDecl`

are stored and manipulated as ASTs, a context refers to a node (descriptor-node) within an AST.

2. `getEnclContext`, given a context (block), returns the enclosing context (block) i.e. in our example, the block corresponding to the body of `Spreadsheet`.

3. `loadPatternDecl` inputs, as text, a Beta pattern declaration such as the `MultiMediaCell`. It could get this text from the user, a file, another process, etc.

4. `AddDecl` then installs the declaration into the body of `Spreadsheet` and then pre-processes it. Once installed, this pattern declaration is available to subsequent pattern declarations that are installed within the scope of this declaration.

   In order to do this, it (see Figure 3.3):

   (a) parses the new declaration, into a bare (without symbol-table info) AST (*new-AST*).

   (b) uses the context information to load the AST of the spreadsheet application (*host-AST*) and locate the `Spreadsheet`-body block node within it.

   (c) installs new-AST into host-AST within that block.

   (d) type checks new-AST as if it appears in that block. If the new AST is type correct, it is updated with symbol-table information.

   (e) generates the runtime object corresponding to the new pattern declaration.[3] Such an interpreter-generated prototype is indistinguishable in structure and interface from compiler-generated ones.

[3] this is called a prototype in Beta terminology; other common terms are template (Simula) or class descriptor (Eiffel).

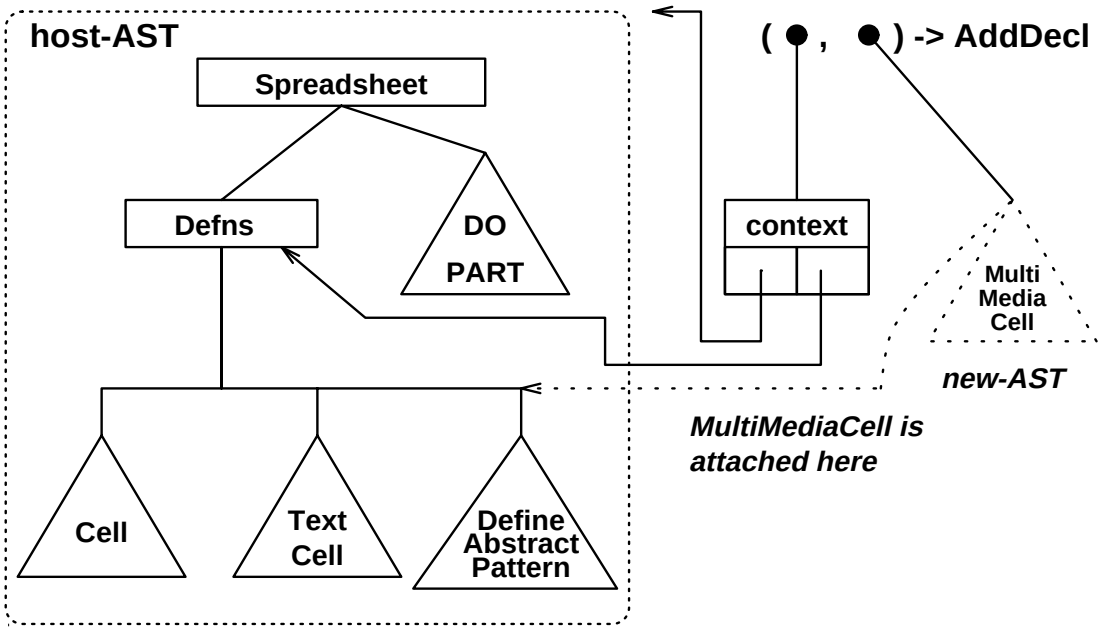


Figure 3.3: Illustrating the details of `AddDecl`

### Contexts

To better understand contexts, observe that the information necessary in order to process a new pattern declaration at a given point in the source program is:

1. The set of source-level symbols visible at that point along with information about their types, block levels, etc.
2. The block level of that point.

Instead of passing all of this to `AddDecl`, we pass it a context which comprises of:

1. name of file containing relevant part of source program AST; this AST is decorated with symbol-table information.
2. an ID which denotes a node within this AST.

Given this, `AddDecl` can derive the necessary information. In our example, `getCurrentContext` yields a context denoting the block corresponding to the body of `DefineAbstractPattern`. In the current Beta runtime system, given an object instance, it is possible to access the runtime representation of its pattern (class), and from there determine the AST node corresponding to the declaration of the pattern. This is the basis for our implementation of `getCurrentContext`.

#### Extensions not Modifications

It is important to note that `AddDecl` doesn't replace an existing compiled declaration of the same name. In other words, loading a new declaration for `TextCell` in the above example would not have the effect of replacing the old compiled `TextCell` declaration. We have devised a method to support such a replacement operation, especially in the case when the signatures of the replacement and the original match. This could be supported by adding another pattern `replaceDecl` to the API. We are still investigating various issues in supporting such an operation and hence don't report its details here. We do not, however, intend `AddDecl` to have such semantics. If we were to load a new declaration for `TextCell` using `AddDecl`, it would override the old declaration only for subsequent declarations made within its scope. Old users of `TextCell` will continue to use the old version. This semantics is in conformance with static binding semantics.

`AddDecl` can add new pattern declarations, but due to the rules of static binding, these are not visible to the compiled code. This is the reason why we have the second function — `MakeDeclExecutable` — in the API.

### 3.3.2 MakeDeclExecutable

This function allows the host-application to execute (instantiate/invoke) patterns in addition to declaring them. Presented below is the spreadsheet application with the implementation of the command Define New Cell; it illustrates the use of `MakeDeclExecutable`. Also shown is `CreateDynamicCell` which illustrates how one uses the return value of `MakeDeclExecutable`. The Create command would do something similar to allow users to create video cells. We assume that the application has been extended with the declaration of `MultiMediaCell`.

```
Spreadsheet :
(#  Cell : (# ... #);
    TextCell : Cell (# ... #);
    DefineAbstractPattern : (# ... #);
    DynamicCell : ##Cell;                          (* pattern variable *)
    DefineNewCell :
    (#  spreadSheetBody : @context;
        spreadSheetObj : ^Object;
        decl : @text;
     DO getCurrentContext -> getEnclContext -> spreadSheetBody;
        getCurrentObject -> getOrigin -> spreadSheetObj[];
        loadNewCellTypeDecl -> decl;
```

```
        (spreadSheetBody, spreadSheetObj[], decl)          (* assign to *)
            -> MakeDeclExecutale -> DynamicCell##;  (* pattern variable *)
    #);  (* DefineNewCell *)
    CreateDynamicCell :
    (#
     DO ... DynamicCell ...                (* execute a pattern variable *)
    #)  (* CreateDynamicCell *)
#)  (* Spreadsheet *)
```

An example of a declaration that can be processed in this way is:

```
VideoCell : MultiMediaCell (# ... #)
```

The pattern `DefineNewCell` encapsulates the functionality of the Define New Cell command. It differs from `DefineAbstractPattern` in that it calls `MakeDeclExecutable` which takes an object as an argument; this is in addition to the context and declaration arguments, both of which have the same purpose and interpretation as in the case of `AddDecl`. The object argument is an instance of the block denoted by the context. Its need becomes evident once we examine the return value of `MakeDeclExecutable`.

The value returned by `MakeDeclExecutable` can be thought of as a pattern-closure i.e. a value which denotes a pattern, may be passed as a parameter, returned as a result, stored in a variable, or used to execute the pattern. In a procedural language, its counterpart would be a procedure pointer, while in a functional language its counterpart would be a function-closure.

For a pattern P, its closure must have an environment link; this is true of closures in functional languages. This environment link is used, at run-time, to resolve names not local to P. In the Beta implementation, such an environment link is simply a reference to an instance of the block with lexically encloses P. The object argument to `MakeDeclExecutable` is used as an environment link while building the pattern-closure. The context argument specifies the block in which P should be processed; thus, this block lexically encloses P. Hence, the object argument should be an instance of this block.

In Beta terminology such a pattern-closure is called a *structure value*. It may be instantiated and executed like a pattern and used almost any place a pattern may be used. In the Beta implementation, a pattern's structure value comprises of a reference to the pattern's prototype and the environment link. Structure values are stored in variables known as *pattern variables*. These concepts are illustrated in Section A.1 in a setting independent of the interpreter. The reader unfamiliar with these concepts in Beta is advised to take a cursory glance at that brief section.

Returning to the example, observe the pattern variable called `DynamicCell`. The return value of `MakeDeclExecutable` is stored in this pattern variable. `DynamicCell` may then be used to execute the pattern denoted by the structure value within it. In the example, `CreateDynamicCell` uses `DynamicCell` to create an instance of the dynamically defined pattern `VideoCell`.

To summarize, using `MakeDeclExecutable` to process a pattern requires (in addition to `AddDecl`) that one pass it an object reference corresponding to the block denoted by the context argument. The return value is a structure value which must be stored in a pattern variable (which must be declared appropriately). The pattern variable can then be used to execute the pattern (from possibly other parts of one's application).

As Beta patterns are the unifying construct for classes, procedures, types, etc., a structure value is capable of denoting all of these. By allowing `MakeDeclExecutable` to return structure values, we get a *simple* but *general* mechanism for embedding new declarations into compiled programs; *simple* because the `MakeDeclExecutable` always returns a structure value denoting a pattern, *general* because everything in Beta can be encapsulated in a pattern.

To summarize, `AddDecl` adds a new declaration to the specified context. This declaration becomes available, within its scope, to other declarations made using `AddDecl` or `MakeDeclExecutable`. `MakeDeclExecutable` adds the declaration to the specified context, like `AddDecl`, but also returns a structure value which may be executed. This pair of functions is sufficient for creating a wide variety of dynamic extensions and for building a variety of interesting applications.

### 3.3.3 Implementation Issues

#### Plugging the type-checking loophole

Type checking occurrences of `AddDecl` is simple; no special treatment is required. But `MakeDeclExecutable` is a little more interesting. We have declared it to return a structure value of the most general type:

```
MakeDeclExecutable : (context * originObject * declText) -> ##Object
```

Given this type, the Beta type-checker will allow the return value of `MakeDeclExecutable` to be assigned to pattern variables of type at least `Object`. `Object` is the root of the class hierarchy in Beta. Hence, all pattern variables, regardless of declared type, are of type at least `Object`. Therefore, the value returned by `MakeDeclExecutable` may be assigned to

a pattern variable of *any* type. This is certainly not acceptable as one could, for example, assign a `VideoCell` structure value to a pattern variable of type `Text`.

This problem necessitates the use of a runtime type-check. The compiler has to be customized to generate such a check during code generation. The runtime check should ensure that the pattern of the structure value returned by `MakeDeclExecutable` is a subclass of the pattern variable to which it is assigned.

### Access to compiled machine code

If the interpreter comes across a pattern application where the corresponding declaration is compiled, it resolves the application by locating the compiled machine code rather than interpreting the declaration. In a Beta executable, every compiled pattern has a prototype. In addition, each of these prototypes has an assembly-level name. It is possible to determine this assembly-level name from the AST node which declares the pattern. So, to locate the prototype of a compiled pattern declaration, the interpreter accesses the declaration node in the AST, gets the corresponding assembly level symbol-name, and reads the corresponding memory address from the linker-generated symbol-table stored within the executable.

The prototype of a pattern is all that's needed to create an instance of a pattern or to execute it. Hence, by getting access to the prototype, the interpreter is able to support creation and execution of compiled patterns. To transfer parameters into a pattern instance before executing it, and to get results out it, the interpreter also needs the signature of the pattern. For this it uses the pattern declaration node in the AST.

Our method for determining the addresses of compiled prototypes works because prototypes are non-relocatable. Should they be relocatable, one would have to do some additional bookkeeping to keep the executable's symbol-table consistent.

### Interpreter-generated prototypes

We stated earlier that interpreter-generated prototypes are identical in structure and interface to compiler generated ones. This property enables us to return interpreter-generated prototypes to compiled code without any problems. These prototypes are illustrated by the following abstract example:

```
COMPILED:                                    INTERPRETED:
P : (# V :< (# DO ... INNER; ... #);         Q : P (# V ::< (# DO ... #);
     DO ... V; ...                                 DO ...
       INNER;                                      #);
     #);
```

The pattern P is declared with a virtual pattern V. In P's body, V is executed; then INNER is called to transfer control to the specialization of P. The virtual V also calls INNER to transfer control to its specialization.

Q is declared as a specialization (subclass) of P. It binds the virtual V further.[4] For the sake of this discussion, we will refer to V declared in P and Q as Vp and Vq respectively. Now, suppose the compiled code also does the following:

```
(* load defn of Q *)
(Definition of Q, Q's context) -> MakeDeclExecutable -> X##;
X;                                                    (* execute Q *)
```

Figure 3.4 illustrates the scenario after an instance of Q has been created via X. This instance is created by compiled code using the structure value returned by MakeDeclExecutable. The Q instance has a reference to its prototype. This prototype is interpreter generated. It has a table with references to Q's body and P's body; the bodies are machine code. Q's body is interpreter generated and is really some "glue" which transfers control to the interpreter core (arrow 7). There is also a virtual-table with an entry for each virtual in Q. This points to the prototype for Vq. This prototype is also interpreter generated. It is similar in structure to Q's prototype except that it doesn't have any virtuals.

1. Q is executed (via X) by transferring control to the body of P (arrow 1).[5]

   (a) When P executes V, V's prototype is accessed from the current object's (Q) prototype's (Q) virtual-table (arrow 2). This is used to create an instance of Vq and control is transferred to the body of Vp(arrow 3).

   (b) When the INNER is encountered, control flows to the body of Vq (arrow 4). This is interpreter-generated machine code; it transfers control to the interpreter (arrow 5). The interpreter is provided with the current object (Vq instance; not shown in figure) and a reference to the AST to be interpreted (i.e. Vq's AST).

[4] The further binding is a subclass of V declared in P.

[5] In Beta, execution begins in the top of the superclass chain and travels down to specializations via INNER statements.

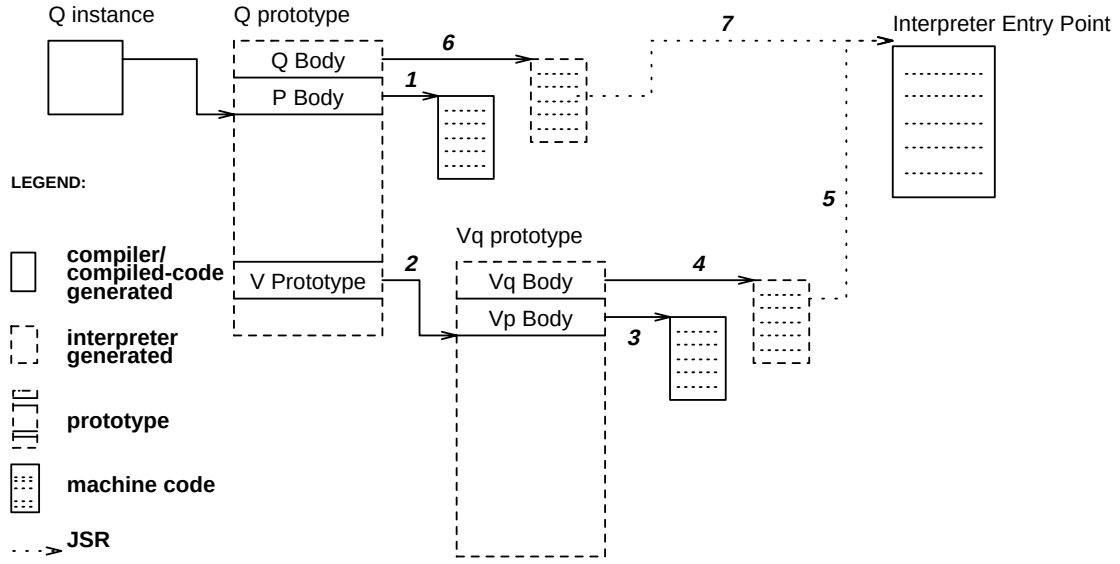


Figure 3.4: Interpreter-generated prototypes

2. Control first returns to the body of `Vp` and then to the body of `P` (from where `V` was executed).

3. When the `INNER` is encountered, control flows to `Q`'s body (arrow 6) which again transfers control to the interpreter (arrow 7), this time with the `Q` instance and a reference to `Q`'s AST.

For more information about the Beta run-time system and compilation techniques see [Mad93].

## 3.4 Other Applications of the Embeddable Interpreter

As has been illustrated by the spreadsheet example, the embeddable interpreter may be used to build dynamically extensible systems. The generality of the interpreter API in combination with the object-oriented features of Beta (such as virtual patterns) provides for a powerful framework for writing dynamically extensible systems. The related report [Mal94] discusses extensibility in Beta and other languages. Sandvad [NS90] also discusses the various tailorability aspects of Beta programs, although in a static context. With the embeddable interpreter, one could extend all of those ideas to work dynamically.

In addition to dynamically extensible systems, there are other applications of such an embeddable interpreter. One obvious application is an interactive development environment

for Beta programs; another is in a source-level debugger for Beta programs. We outline these two applications in this section.

We have also explored the role of the embeddable interpreter, in making incremental a rapid prototyping tool called the Application Builder (*ApplBuilder*) [GHT91]. The ApplBuilder, among other things, allows users to create user-interfaces by direct manipulation techniques. It generates Beta code which must be compiled before the interface can be tested. Should the interface need editing, the generated code may be reloaded into ApplBuilder and edited. We have devised a scheme which allows the edited version to be executed without regenerating or recompiling a new executable. This is accomplished by interpreting the changes in the context of the original compiled application. The details of this are too lengthy to report here.

### 3.4.1 Interactive Development Environment

The interpreter can be used to build an ordinary interactive development environment where one can interactively interpret "fragments" of Beta code. The following is a simplified "listener" loop for Beta:

```
ListenerLoop :
(#  X : ##Object;
    <<PERVASIVE ENVIRONMENT DEFINITIONS>>;
 DO loop :
      (if (Command)
         "Define" then
             (getCurrentContext, getDecl) -> AddDecl;
         "Execute" then
             (getCurrentContext, getCurrentObject, getDecl)
                 -> MakeDeclExecutable -> X##;
             X;
         "Quit" then
             leave loop;
       restart loop;
      if)
#)
```

When it gets a "Define" command, it obtains a pattern declaration from the user and calls `AddDecl`. The interpretation-context is the body of `ListenerLoop`. This contains declarations of all the pervasive patterns (the patterns available upon startup of the listener) and the declarations added via `AddDecl` and `MakeDeclExecutable` since startup.

The pattern declaration is interpreted in this context and also added to the context. When an "Execute" command is received, the declaration is processed via `MakeDeclExecutable` which returns the structure value of the pattern. This is stored in `X` and executed.

In this example, all declarations are processed in, and added to, the body of `ListenerLoop`. One cannot, for example, add a new pattern-declaration (method) *within* a pattern declared in the pervasive environment. The pervasive environment has, for example, the declaration of a pattern called `Text`. With the given listener loop, it is impossible to add a declaration into the body of `Text`. Enabling this would allow a user to extend `Text` with a new method, such that subsequent declarations could use this extended `Text`.

The API function `AddDecl` is sufficient to support this feature, for it takes a context as an argument. If called with context equal to the body of `Text`, it will add the declaration there. The problem lies in getting this context information from the user and providing it to `AddDecl` in the desired form. With a more sophisticated user-interface, such as an AST browser, which could translate a user mouse-click on a node into a context value denoting that node, we could support this feature. The Beta system has a syntax-directed editor called sif [MI93] which allows one to edit Beta program ASTs. Embedding the interpreter into sif would allow us to support the addition of declarations in arbitrary contexts.

### 3.4.2 Debugger

Source level debuggers which allow one to execute arbitrary code at a break point, using the break point context, are useful and powerful tools. The embeddable interpreter can be used to support such facilities in a debugger. Let the interpreter be part of the debugged application. To execute a pattern declaration at a breakpoint, the debugger can invoke `MakeDeclExecutable` and then execute the returned structure value. In order to invoke `MakeDeclExecutable`, it needs:

1. the context corresponding to the breakpoint. This is really block in the source program where we are stopped. The Beta debugger (valhalla [MI92a]) is AST based and is capable of providing this context information in the form required by `MakeDeclExecutable`.

2. the object corresponding to the breakpoint i.e. the object within which we are stopped. Most debuggers are capable of displaying the current object (frame) and hence this should not be a problem.

3. the pattern declaration to be executed.

As all of this information is available for any breakpoint, `MakeDeclExecutable` is invocable at any breakpoint.

Should we have a function, in the API, that allows the replacement of compiled pattern-declarations by interpreted ones (`replaceDecl`) (see Section 3.3.1), we could even support the patching of a program during debugging. With this capability, the debugger would be much like an incremental execution environment.

## 3.5 Applicability to Other Languages

The interpreter API and its supporting interpreter core, dictate a set of requirements on the language and implementation. The previous sections have indirectly touched upon these requirements in the context of Beta. In order to make these ideas more widely applicable, we list these requirements in a language independent manner. We also explore the possibilities of having such an embeddable interpreter for Simula [BDMN73], Eiffel [Mey88], and C++ [ES90]. These languages belong to the family of compiled, statically- and strongly-typed languages, and hence stand to benefit from having such an embeddable interpreter.

The requirements include the following:

- *Classes as Values.* For the API to include a function like `MakeDeclExecutable`, the language must support the notion of *class-values* (structure values in Beta). It is *not* necessary to have classes be full-fledged objects whose behavior can be specified in the metaclass as is the case in Smalltalk [GR83]. Class-values should be just "black-box" values (like closures, their internals should be of no interest at the language level); it should be possible to pass them as parameters and store them in variables, in addition to using them just as we would use a statically declared class i.e. for creating instances. In Beta, structure values are surprisingly simple objects.

  As an alternative to returning class-values, the function `MakeDeclExecutable` could simply execute the class and return the result of the execution as its result. Although this approach accomplishes our goal, it isn't as elegant as the one using class-values. With the class-value approach, we get an almost seamless boundary between compiled and interpreted code. Once an interpreted pattern declaration has been processed into a structure value, the compiled code doesn't need to be aware of the creator of the structure value; it may use it just as it would a compiler-generated structure value.

- *Classes at Runtime.* Our implementation relies on the fact that there exists at run-time, for each class, a data structure providing information about the class.

These data structures are called prototypes in Beta, templates in Simula, and class descriptors in Eiffel. Note that these are not the same as class-values; this is a run-time data structure, while a class-value is a language level notion.

- *Context Specification.* We rely on being able to uniquely address any block in the source program (even if it's in multiple files). In addition, it must be possible to get such address information about a program from within that program. The functions `getCurrentContext` and `getEnclContext` provide such information in our implementation. This is not much of an issue in languages without block structure.

- *Object Specification.* If we can compute the address of a block, then we should also be able to get a handle on its corresponding instance. `getCurrentObject` and `getOrigin` are the counterparts of the context specification functions in our implementation. In a language without block structure, the origin, if maintained, would be a fixed (root) object.

- *Symbol Table.* It is imperative that the interpreter be able to map source level names of compiled classes into run-time memory addresses of the corresponding prototypes (templates/class descriptors).

- *Type Checking.* It must be possible to determine the type of a structure value at run-time. This is the basis for the dynamic type-check.

Beta resembles Simula in many ways; those relevant to this discussion include block-structure, and typing. The primary difference (for this discussion) is that Simula doesn't have the notion of class-values. This can be overcome by either introducing such a concept or by using the alternative approach which doesn't require class-values. Simula also doesn't have patterns as the unifying concept for classes, types, functions, procedures, etc. As a result, if we want dynamic interpretation of syntactic constructs of granularity finer than class, we will have to support it explicitly. In other words, if we want to be able to dynamically interpret procedures (to add it to a class, for example), we have to have an API function for procedures as well as one for classes. In the Lund Software system for standard Simula [Lun92], it is possible to map a source-level class name into the address of its corresponding prototype [LM84, Hed93].

It should be possible to build an embeddable interpreter for Eiffel, without sacrificing the safety of programs that use it. Class-values are not present in Eiffel; hence, the alternative approach as described for Simula can be used. According to the implementation description in [Mey88], class descriptors, the data structures representing classes, are present at run-time. It is not clear if one can map a class name into the address of its class descriptor, but one would expect this to be possible. Context and object specification is

much simpler as there are no nested classes. Like Simula, class, procedure, and function declarations are not unified into a single abstraction. With respect to our concerns here, C++ [ES90] falls into the same category as Eiffel.

## 3.6 Related Work

To the best of our knowledge, statically-compiled object-oriented languages like Eiffel [Mey88] and C++ [ES90] don't have such embeddable interpreters. To obtain dynamic extensibility in an application written in these languages, developers have to resort to designing an interpreted extensibility language with a predefined set of functions which access the underlying application's functionality. Another approach used in [ZC92] is to embed an interpreter for an interpreted language like Scheme [Bet89] into the application. The authors of [ZC92] state "The Scheme interpreter is used mainly to invoke C++ functions and this might seem to be an overkill." Using Scheme, or another such language, to extend an object-oriented application cannot possibly allow very general extensions to be written. Extending Beta applications, using Beta, one can utilize the application's object-oriented model to the maximum in writing the extensions.

Embeddable interpreters are most common in the Lisp world. GNU Emacs has a lisp interpreter embedded within it [Sta85]. The interpreter is available to emacs lisp programs as the function `eval`, which is documented as follows:

> eval : Evaluate FORM and return its value.

where both FORM and value are simply S-expressions. Our interpreter has all the flexibility of such an `eval`, but in the context of a block-structured, statically-scoped, statically-bound, statically-typed, object-oriented language. Block-structure and static-scoping force us to introduce interpretation-contexts. To overcome static-binding we introduce pattern variables and the related structure values. These concepts also help solving the problem of what should be returned by the interpreter. In Beta, programs are not data; this problem is solved by using ASTs. Strong typing is enforced by a run-time type check. For an interesting discussion on strong typing in object-oriented languages, see [MMMP90]. The idea of pattern variables in Beta was originally introduced in [AFO89]. In [AF89], Agesen and Frølund present a mechanism for building extensible systems using dynamic linking and loading.

CLOS [Kee89] also provides access to its interpreter (`eval`) its compiler (`compile`) and binding environment (`boundp` and `makunbound`). From our point of view, this is similar to the Emacs lisp capabilities, except that, here we are in an object-oriented setting.

Due to the dynamic nature of Smalltalk [GR83], dynamically extensible applications can easily be implemented in it. Smalltalk makes this possible by providing its compiler as just another object to which "eval" messages may be sent. The source code to be evaluated is provided as a string, while the context is provided in the form of dictionaries. The returned value, an instance of `CompiledMethod` can then be manipulated by the user program; it can, for example, be stored in some dictionary where it would influence the behavior of the rest of the program. This flexibility in Smalltalk comes, however, at the expense of safety, efficiency, and readability. One only discovers a problem with the extension when the extension gets executed. Also, Beta's table-driven method lookup provides constant time access for methods (even for interpreted ones). This is in contrast with Smalltalk's dynamic method lookup technique which depends on the length of the superclass chain (caching techniques help a little here).

It is also possible to have an embeddable incremental compiler. There is no conceptual problem in replacing the interpreter core with an incremental compiler. The API should remain the same. In fact, the interpreter is already generating machine code "glue" which branches to the interpreter core. Instead of this, it could just as well generate all the machine code. The Mjølner Orm system [Mag93] has an incremental compiler for Simula. Fine discussions of incremental compilation problems are presented in [Hed92, HM87, HM86].

A number of commercial systems use dynamic linking and loading as a basis for extensibility. They are generally able to load an extension and thus enhance/modify their functionality; they generally don't support the definition of the extension. In such systems, extensions are generally not meant to be user defined, and in cases where they are, they are rarely meant to be defined interactively. Our approach, in addition to supporting the loading and linking of extensions, also supports their dynamic and interactive definition. Our approach allows for the development of applications in which the distinction between extending and using is blurred.

Another approach to supporting extensibility is to have a meta-level architecture as in the metaobject protocol for CLOS [KdRB91]. Given such a metaobject protocol for Beta, we would be able to create new classes and methods dynamically by creating the appropriate metaobjects. But, in order to construct the "raw materials" (e.g. the machine code of a method body) needed to create the metaobjects, from the source code provided by the user, we would need a processor (interpreter) like the one we have described here. So, adding a metaobject layer doesn't preclude the need for an embeddable interpreter like the one we have described. A metaobject layer would complement the ideas we have presented here; the interpreter could, for example, be used to modify metaobjects of existing classes. Furthermore, introspection and analysis could prove useful in a user-tailorable system. In our approach, structure values returned by the interpreter, can

be thought of as metavalues; API functions like `AddDecl`, `getCurrentContext`, etc. are reflective in that they allow us to operate on the program itself.

Extensibility in C++ with a meta-level architecture is presented in [CM93]. They present a language called OpenC++ in which classes and methods may be declared reflective. Reflective classes have metaobjects; these can be used to extend or change the semantics of method calls. These facilities, while potentially providing the necessary infrastructure in C++, for building an embeddable interpreter like the one we have described, don't preclude the need for an embeddable interpreter.

## 3.7 Current Status, Performance Issues, Future Work

The interpreter has been developed on an HP-UX series 400 machine. There are plans to make it available on other HP architectures, SUN, and Macintosh; an intermediate version during development was able to run on a Macintosh. As the majority of the system is written in Beta, we don't anticipate many porting problems.

We have successfully embedded the interpreter into a simple listener loop (Section 3.4.1) to create an interactive development environment. On a standard Beta demonstration program which creates a simple database, populates it with entries, and displays the entries under various categories, the interpreter performed as follows (all times in seconds):

| Interpreted | | | Compiled |
|---|---|---|---|
| Definition | Execution | Total | Execution |
| 0.62 | 5.22 | 5.84 | 0.20 |

The demonstration program comprises of 284 lines of Beta code. Of the 5.84 secs taken by the interpreter, 0.10 secs (1.7% of total) are spent on loading ASTs (of the development environment and the demonstration program). These measurements are for the first *unoptimized* version of the interpreter.

A potential area of concern, from a performance standpoint, is the loading of the assembly-level symbols from the executable. This happens *only once* for every instance of the interpreter, when the interpreter is initialized; it takes 8.34 secs for the listener loop application which has 9295 symbols. Note that the number of symbols depends on the application into which the interpreter is embedded; an application with more classes linked in will have more symbols.

We are also experimenting with embedding the interpreter into Beta applications with graphical user-interfaces, our goal being the development of a framework for extensible user-interfaces. At present we have embedded the interpreter into an X-windows based text editor. To get an idea of the size overhead of the interpreter, we measured the size of the editor without the interpreter (2.07MB) and with the interpreter (3.42MB), an increase of 1.35MB or 65%. In the case of the editor, the number of symbols is 18088.

We plan next to embed the interpreter into an X-windows based drawing tool written in Beta. The interpreter will be used to make extensible various components of the application e.g. we will allow users to define new types of graphical objects.

On another course, we plan to embed the interpreter into *sif* (the syntax-directed editor) [MI93]. We envision users being able to interpret parts of the program they are editing by specifying (interactively) the context in which to execute it. Integration of the interpreter with *valhalla* [MI92a], the Beta source-level debugger is also forthcoming.

## Acknowledgements

This work has been generously supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. I would like to thank Prof. Ole Lehrmann Madsen for the numerous discussions on the material presented here and for countless personal consultations on the secrets of the Beta implementation. I thank Görel Hedin and Charles Lakos for their valuable comments on earlier versions of this paper. Thanks to the anonymous referees; their comments helped improve the paper. Randall Trigg and Kaj Grønbæk stimulated me with the problem of making the ApplBuilder incremental; I thank them. Furthermore, I appreciate the help of Elmer Sandvad, Jørgen Nørgaard, Jørgen Lindskov Knudsen, Søren Christensen, Niels Damgaard Hansen, and the folks at Mjølner Informatics. I thank Kurt Jensen for helping me come to Aarhus. And last, but not least, thanks to all the folks working hard at keeping the DeVise project going and the sources that fund the DeVise project.

# Chapter 4

# On the Implementation of an Interpreter for Building Extensible Applications

**Author:** Jawahar Malhotra

**Date:** October 1993

**Publication Information:** Aarhus University Technical Report [Mal93c].

**Abstract.**

This paper describes the implementation of an interpreter for the object-oriented programming language Beta. This interpreter has been designed specifically for constructing extensible applications, an extensible application being a ready-to-run application that has the potential of being extended in its use environment. The paper shows how any aspect of an application can be made extensible by a call to the interpreter. It then traces the call to the interpreter, giving an overview of what happens behind the scenes. In particular, it shows how the interpreter handles extensions in any scope of the original program, how it ensures the soundness of the extended program, and how partly-compiled and partly-interpreted object instances get created. It is not meant to be an exhaustive guide to implementing an interpreter for Beta. It highlights only those aspects of the implementation that concern extensibility.

## 4.1 Introduction

By an extensible application is meant a ready-to-run application which has the potential of being extended in its use environment. Extending the application should *not* require (1) any modification of the original application's source code, (2) any recompilation of the original application's source code, (3) relinking of the entire application from scratch. Furthermore, the extensions must be able to add significantly new functionality to the application, functionality not conceived of at application development time.

An example of such an application is the text editor GNU Emacs [Sta84]. It is a ready-to-run application which is capable of being extended in significant ways using the embedded lisp compiled/interpreter. Making extensions doesn't require the source code of Emacs, leave alone any modifications, recompilations, or relinking. A user simply writes an extension as a lisp function and installs it as a callback. The built in lisp processor interprets this function and incorporates it into its current execution environment. Motivation for the need for extensible applications can be found in [Mal94, Mal93a, Kic92, TMH87].

A goal of our work has been to develop support for building such applications, but in the object-oriented programming language Beta [MMPN93]. Object-oriented languages are inherently suited for writing systems that are extensible — this was one of the reasons for our choice of Beta. Its object-oriented facilities (patterns as a unifying abstraction mechanism, virtual patterns, pattern variables) are well suited for constructing extensible software. Another useful feature of Beta is its block structure and lexical scoping — this, among other things, serves to localize the context that an extension has to deal with. There are several other motivating factors for building extensible applications in a language like Beta. They are not the focus of this paper. They are discussed in detail in [Mal94].

Our approach to building such extensible applications in Beta requires that an interpreter for Beta be embedded into the application. The application is developed in an extensible manner with open points which are capable of accepting extensions. The application invokes the interpreter in order to process extensions in the context of the original application. The extensions are also written in Beta; they may access parts of the original application just as if they had been textually inserted into the source of the original application and compiled. Other alternative approaches to building extensible applications are described in [Mal94].

This paper describes the implementation of the Beta interpreter designed to fulfill these requirements. This is interesting because:

1. The interpreter is able to handle extensions of patterns defined in any lexical scope — even nested ones.

2. The interpreter ensures continued type-safety of the application even after extensions are made.

3. There is a seamless interface between the extensible application and the extensions, even though the extensible application is compiled and the extensions are interpreted.

4. There is an interesting and unusual blend of compiled and interpreted code.

This paper is not an exhaustive guide to implementing an interpreter for Beta. Although it touches upon many implementation issues for Beta, it focuses primarily on how the interpreter processes extensions in the context of the extensible application. It shows how this can be accomplished without extending the runtime-system data-structures used by the compiler. It is not necessary to be familiar with the Beta runtime system in order to understand the ideas presented here — they are introduced when needed.

This document could be used in conjunction with [Mad93] to gain a thorough understanding of the implementation of the interpreter. This would be useful for someone intending to maintain or further develop the interpreter.

Prior knowledge of the Beta programming language, although beneficial, should not be necessary to read this paper. A brief primer is presented in Appendix A. It is recommended reading for those not familiar with Beta.

It is worthwhile to compare and contrast this report with two others written by the same author. The report [Mal94] addresses issues in the construction of extensible systems. It doesn't deal with any implementation issues of the interpreter. The report [Mal93a] documents the application programmer's interface to the Beta interpreter and shows how it can be used to build extensible systems as well as for other purposes. It touches upon implementation issues, but only in a superficial manner. This report, on the other hand describes, in detail, the interesting aspects of the implementation of the interpreter. In order to illustrate the use of the interpreter, and to setup an example, it begins with some material that may overlap with the other two papers. However, even here, there is something new — the interpreter is invoked with a closure instead of a context and an origin object.

The paper begins with an example which illustrates, intuitively, the types of extensibility the interpreter is designed to deal with (Section 4.2). The example is then used to illustrate the most interesting aspects of the implementation (Section 4.3). It concludes with some remarks (Section 4.4) and a discussion of related work (Section 4.5).

## 4.2 The Central Idea

### 4.2.1 Types of Extensions Desired

Say we have an application structured as follows:

```
GraphicalEditor:
(#  Display :
    (#  screen : @screenDesc;                    (* instance variables *)
        Window :                                    (* nested patterns *)
        (#  contents : @bitmap;                  (* instance variables *)
            refresh :< (# do ... contents ...
                                screen ...; INNER #)       (* methods *)
            GraphicalObject : (# ... #)             (* nested patterns *)
            Circle : GraphicalObject (# ... #)
        do  ... contents ... screen ...; INNER
        #)
        cw : ^Window;
    do  ... new(Window) -> cw[]; cw; ...
    #)
    d : ^Display;
do  ... new(Display) -> d[]; d; ...
#)
```

Here the pattern `GraphicalEditor` is meant to be an editor for editing figures. The pattern `Display` is nested within it. It has instance variables: `screen` which describes the screen it should display on, and `cw` which stores a reference to the current window. The pattern `Window` which describes a typical window is nested within `Display`. It has an instance variable: `contents`. It also has a virtual pattern: `refresh`. The pattern `GraphicalObject` is an abstract description of all graphical objects to be handled by this application; `Circle` is a concrete sub-pattern of it. Both their declarations are nested within `Window`.

The `GraphicalEditor` pattern has a do-part; this implies that it may be executed. Executing it will create an instance of it and run the do-part. The do-part will typically setup the overall user interface. During this process it will create an instance of the `Display` pattern and execute it; this is shown in the listing. The `Display` will create one or more `Window` objects and manage them; it will provide the main interaction with the user. The listing shows it creating and executing an instance of `Window`. The `Window` pattern has been designed so that executing it will initialize it. Its do-part is shown to have references to variables: `contents` and `screen`. At times, the `Display` may also call `refresh` on the

`Window` (not shown in the listing). The `INNER` imperative appearing in the do-parts of `Window` and `refresh` is explained later.

The above code is block structured and lexically scoped. It is impossible to create, for example, a `Window` from outside a `Display`. This adds to the clarity of the code by expressing existential dependencies: a window cannot exist without a display. The many other benefits of block structure and lexical scoping, which are not the focus of this paper, are also enjoyed by this application.

There are a number of ways to extend this application; for example, one may want to introduce colored windows, or change the way circles are highlighted when they are dragged. One way to introduce colored windows, is to define a new pattern describing colored windows, say `ColorWindow`, as a sub-pattern of `Window`. Place this definition in the same block as `Window`. Replace all applied occurrences of `Window` by `ColorWindow`, and then recompile the resulting source.[1] The following listing shows this extended program:

```
GraphicalEditor:
(#  Display :
    (#  screen : @screenDesc;                    (* instance variables *)
        Window :                                    (* nested patterns *)
        (#  contents : @bitmap;                  (* instance variables *)
            refresh :< (# do ... contents ...
                                screen ...; INNER #)       (* methods *)
            GraphicalObject : (# ... #)             (* nested patterns *)
            Circle : GraphicalObject (# ... #)
        do  ... contents ... screen ...; INNER
        #)
        ColorWindow : Window
        (#  color : @colorDesc;
            refresh ::< (# do ...contents... screen ... color; INNER #)
        do  ... contents ... screen ... color ...; INNER
        #)
    cw : ^Window;
    do  ... new(ColorWindow) -> cw[]; cw; ...
    #)
do  ... new(Display) -> d[]; d; ...
#)
```

[1] One could also have declared `Window` and `Display` as virtuals in the original implementation, thus allowing the window extension to be installed without modifying the original definition. I have deliberately not chosen this approach so that I can demonstrate the use of the interpreter.

In this case, the `Display` pattern is shown creating and executing an instance of a `ColorWindow`. When an instance of `ColorWindow` is executed, the do-part of `Window` is executed until the `INNER` imperative is encountered. Then, control is transferred to the do-part of `ColorWindow`. The `INNER` here is like a no-operation, as there are no further specializations. Similar comments apply to the do-part of the `refresh` pattern.

The goal of this section has been to demonstrate the types of extensions (e.g. `ColorWindow`) the interpreter has been designed to support.

### 4.2.2 Using the Interpreter

The interpreter has been designed as a tool to support the above types of extensions. It is meant to be used to write an application, such as the editor above, with a well-defined set of open points — points that are open for extension. Such an open or tailorable application has built into it, the ability to load the source code for extensions such as `ColorWindow` and extend itself accordingly. Furthermore, these extensions may happen dynamically and without any recompilation or relinking of the original application.

In order to enable the editor to accept extensions such as `ColorWindow`, it must be rewritten as follows:

```
GraphicalEditor:
(#  Display :
    (#  screen : @screenDesc;                    (* instance variables *)
        Window :                                    (* nested patterns *)
        (#  contents : @bitmap;                  (* instance variables *)
            refresh :< (# do ... contents ...
                                screen ...; INNER #)       (* methods *)
            GraphicalObject : (# ... #)             (* nested patterns *)
            Circle : GraphicalObject (# ... #)
        do  ... contents ... screen ...; INNER
        #)
        cw : ^Window;
        windowExtension: ^text;
        WindowX : ##Window;
        extendWindow :
        (#
        do  inputFromUser -> windowExtension;         (* textual input *)
            (Window##, windowExtension) -> interpret -> WindowX##;
        #)
    do  ... new(WindowX) -> cw[]; cw; ...
```

```
        #)
        d : ^Display;
        ... new(Display) -> d[]; ... d.extendWindow ...
    #)
```

This listing differs from the original in that it declares a pattern variable[2] `WindowX` (a variable which holds pattern closures[3], as opposed to instances of patterns). The pattern variable `WindowX` is declared of type `Window`. This means that it can hold the `Window` pattern or any sub-pattern of it. Also, references to the pattern `Window` have been replaced by references to the pattern variable `WindowX` — thus allowing the pattern denotable by these references to be dynamically changed.

A pattern called `extendWindow` has been declared within `Display`; calling it on a display object extends the definition of `Window`. `windowExtension` denotes the source code for the new definition; its value can be specified dynamically, as is exemplified by the `inputFromUser` imperative. Any source code which defines a sub pattern of `Window` can be used here; as an example, consider `ColorWindow` from the previous listing:

```
ColorWindow : Window
(#  color : @colorDesc;
    refresh ::< (# do ...contents... screen ... color; INNER #)
do  ... contents ... screen ... color ...; INNER
#)
```

Within `extendWindow`, the interpreter is invoked with this source code and a closure for the `Window` pattern (i.e. `Window##`). By passing the pattern closure for `Window`, the user indicates that she/he would like the extension to be processed in the same lexical scope as `Window`. Thus, the interpreter processes the extension as if it appeared within `Display`. This scope information is used by the interpreter to determine which name references (e.g. `Window`, `contents`, `screen`) in the interpreted code are legal, as well as how to resolve them (i.e. map a reference to a memory address).

An intuitive way to explain the arguments with which the interpreter should be invoked is: call the interpreter with the source code of the extension pattern and the pattern closure of the pattern to be extended. The interpreter returns the pattern closure of the extension pattern.[4]

[2]Pattern variables are a standard feature of Beta

[3]These are called structure values in Beta; pattern closures have a more intuitive connotation for this discussion and hence are used here

[4]This is a natural extension of the standard way in which pattern closures are created and assigned in Beta: the statement `Window## -> WindowX##`, constructs a pattern closure for a pre-defined `Window`

Returning to the editor example, when this application creates a new display object and calls `extendWindow` on it, the display gets extended with a new user-defined definition of window, such as `ColorWindow`. In other words, the definition of an extension to `Window` gets loaded into the application without any need for recompilation, relinking, or a restart of the original application.

### 4.2.3 The contribution

This paper discusses the implementation of an interpreter designed to accomplish the above-described tasks. What makes this interesting are a number of things:

1. The ability to handle extensions of patterns defined in any lexical scope — even nested ones. Lexical scoping rules are not violated by this; the extension gets access to an environment as per the lexical scoping rules. Lexical scope is provided as an argument to the interpreter. The interpreter can be used to handle extensions of any pattern in any scope e.g. `Circle` in `Window`.

2. Type checking of the extensions and preservation of the type soundness of the original program.

3. The packaging and return of the interpreted pattern as a pattern closure. This results in a seamless interface between the interpreter and the compiled application invoking the interpreter.

4. An interesting and unusual blend of compiled and interpreted code.

## 4.3 Implementation Details

The signature of the interpreter can be described as follows:[5]

```
interp: (closure of to-be-extended pattern *
         source of extension pattern) -> closure of extension pattern
```

pattern and stores it in the pattern variable `WindowX`, while the statement `(Window##, extension) -> interp -> WindowX##` creates a pattern closure for an extended version of `Window` and stores it in the pattern variable `Window`; here `extension` is the extension in source code form.

[5] This corresponds to `MakeDeclExecutable` ([Mal93a]) with the context and origin objects replaced by the pattern closure of the to-be-extended pattern. The two approaches are semantically equivalent; the pattern closure approach is conceptually cleaner.

The term *to-be-extended pattern* denotes the pattern being extended while *extension pattern* denotes the pattern defined as an extension of the to-be-extended pattern.

The implementation of the interpreter is illustrated using the example of the previous section. There, within `extendWindow`, the interpreter is invoked as follows:

```
(Window##, windowExtension) -> interpret -> WindowX##;
```

It is assumed that `windowExtension` contains the source code for `ColorWindow` (reproduced here for easy reference):

```
ColorWindow : Window
(#  color : @colorDesc;
    refresh ::< (# do ...contents... screen ... color; INNER #)
do  ... contents ... screen ... color ...; INNER
#)
```

The implementation is described in two stages:

- construction of the closure of the extension pattern i.e. the value returned by the interpreter.
- use of this closure via the pattern variable `WindowX`.

Figure 4.1 depicts, partially, the state of the computation before the construction of the extension closure is begun. It shows the closure of the to-be-extended pattern `Window`.

A word about the type safety of the call to the interpreter. For every call to the interpreter:

```
(Q##, extension) -> interp -> X##        where X : ##P
```

it must be ensured that `Q` is a sub-pattern of `P`; this can be done statically. During interpretation of the extension source code, it will be verified that the pattern declared in the extension is a sub-pattern of `Q`, thus ensuring that the extension pattern is a sub-pattern of `P`, and therefore making it safe to assign the extension pattern closure to `X`.

### 4.3.1 Building the closure of the extension pattern

A pattern closure comprises of two parts: a runtime descriptor of the pattern and an environment pointer. This runtime descriptor of the pattern, called a *prototype* in Beta

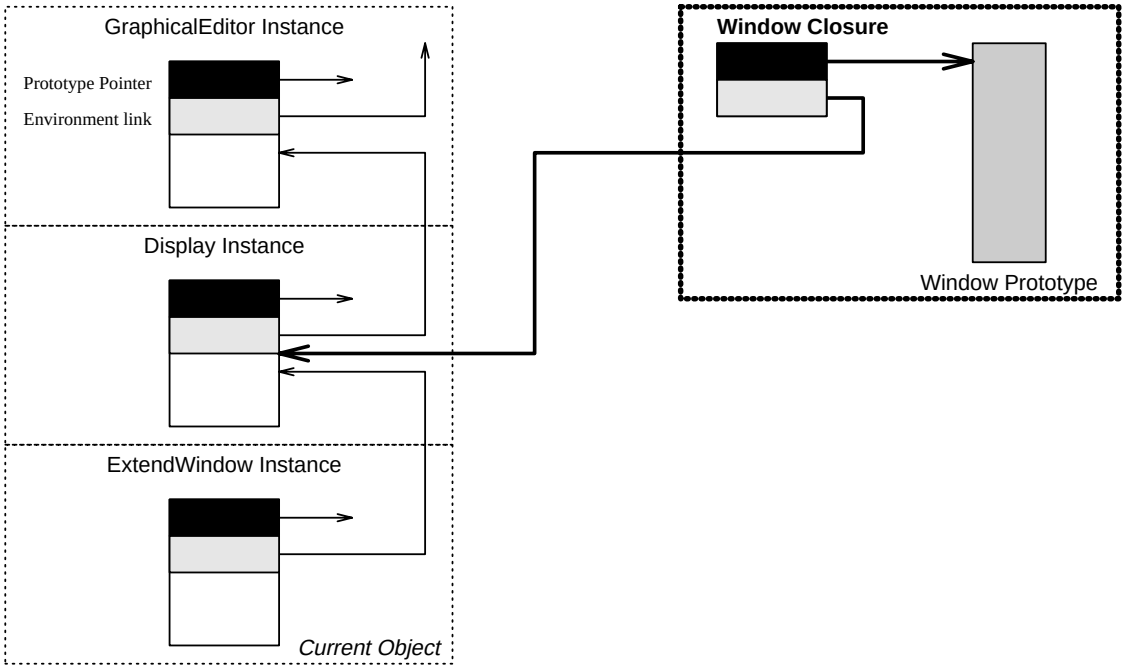


Figure 4.1: The to-be-extended pattern closure

runtime system terminology, is an encoding of the pattern suitable for executing or instantiating it. It comprises of many parts, some of which are discussed in the sequel. The environment pointer is a pointer to an object which is an instance of the pattern enclosing this pattern. This is used to resolve non-local name references. As an example, the closure for the `Window` pattern, built using the `Window##` construct, would have an instance of `Display` as its environment object (Figure 4.1). This would be used to handle non-local references such as `screen` from within `Window`.

In order to construct the closure for the extension pattern, it is necessary to (1) build a prototype for the extension pattern, and (2) package this prototype with the correct environment object.

### Building the prototype

**The Symbol Table.** Before building the prototype for `ColorWindow`, the interpreter needs to type-check its source code. During this process, the symbols used in the definition of `ColorWindow` are resolved — their corresponding symbol-table entries are located. The user of the interpreter has indicated, by passing the pattern closure of the to-be-extended pattern, the lexical scope in which she/he wants this source code to be processed. The check must be performed as if the source code appeared in this lexical scope.

In order to do this, the interpreter must have access to the symbol table of the program being extended. In order to get the effect of processing the extension in this scope, the

interpreter must be able to get access to all symbol-table entries visible in this scope. Furthermore, it must be able to restrict the access, of the extension, to these and only these entries.

These requirements mean that the symbol table, generated when the original application was compiled, be available in some form. In addition, it should support the mapping from any scope to the set of symbols visible in that scope. Furthermore, if extensions are to be visible to future extensions, the table must be updatable with new entries.

In the Beta system, programs are stored as Abstract Syntax Trees (ASTs). Symbol-table information is stored as annotations to the nodes of these trees. So, the symbol table is available to the interpreter in the form of annotated ASTs of the original application. Given this representation, it is possible to denote a scope by a node in this tree. Given a node in the tree, it is possible to determine the set of symbols visible from that node. It is worthwhile to note that although the entire source code of the original program is part of the tree and hence, is available to the interpreter, only the symbol-table information is actually used. One could imagine trimming these ASTs so that only symbol-table information remained. These trimmed trees would be adequate for installing extensions. In fact, a prototype implementation of these trimmed trees is available. Thus, the original requirement that the source code of the application being extended need not be necessary in order to install extensions, has not been violated.

The extension must be processed in the same scope as that in which the to-be-extended pattern was processed. So, in order to find the extension-processing scope, one must find the to-be-extended pattern's scope. This can be found by examining its pattern closure. This contains the prototype, which contains information identifying a file containing an AST, and a node within this AST. This node is the declaration of the to-be-extended pattern. Its enclosing block is the desired extension-processing scope.

The interpreter loads this AST, locates the node corresponding to the extension-processing scope, parses the extension source into an AST, attaches the extension AST at this node, and then type-checks the extension AST, annotating it with symbol-table information. Attaching the extension AST has the effect of installing the new extension into the symbol table.

In the example, the pattern closure for `Window` is passed as an input argument. It is used to locate the node for `Display` in the AST of the original program. The AST for `ColorWindow` is then attached to the original program's AST as if it appeared within `Display`.

**Type Checking.** Once the extension has been setup in the correct scope, the type checking process performs some standard type-consistency checks. The details of these

checks are beyond the scope of this paper; see [MMMP90] for details. Only some of the results of this process are summarized here:

1. The reference to `Window` as the super pattern of `ColorWindow` gets resolved to the `Window` defined within `Display`.

2. Within the definition of `ColorWindow`, there is a further-binding of `refresh`. This gets resolved to the virtual pattern `refresh` declared within `Window`.

3. Within the further binding of `refresh`, there is a reference to the variable `contents`. This gets resolved to the `contents` variable declared within `Window`. It is also recorded that in order to resolve this reference at runtime, one must follow one environment link to go from the `refresh` instance to the `ColorWindow` instance (`contents` is inherited by `ColorWindow` and hence is part of a `ColorWindow` instance), and access the field at `X` bytes offset. Here, `X` is determined from the symbol-table entry for `contents`.

   This shows how accesses from an interpreted extension to the state defined by its super pattern are handled.

4. The reference to `screen`, also appearing within the `refresh` further binding, gets resolved to the `screen` declared within `Display`. Its runtime resolution involves following two environment links (from a `refresh` instance to a `ColorWindow` instance to a `Display` instance), and then accessing the field at `X` bytes offset. Here `X` is obtained from the symbol-table entry for `screen`.

   This shows how accesses from an interpreted extension to the state defined by its enclosing patterns are handled.

**Building the prototype for `ColorWindow`.** As `ColorWindow` is a specialization of `Window`, its prototype is built using `Window`'s prototype as a starting point. The prototype for `Window` is located by getting its memory address from the symbol-table entry for `Window`. In Beta, this is actually a two-step process: (1) from the symbol-table entry for `Window`, obtain an assembly-level symbol name which identifies the prototype in the executable, (2) using the linker-generated symbol table, which is in the executable, map this name into an address.

At this point, before proceeding with the construction of a prototype for `ColorWindow`, it is possible to complete the type checking of the call to the interpreter (first referred to in the beginning of this section). It remains to be ensured that the extension pattern is indeed a sub-pattern of the to-be-extended pattern. The prototype of the super-pattern of

the extension pattern has been obtained from the symbol table. The prototype of the to-be-extended pattern can be obtained from its pattern closure. Using these prototypes, it is possible to check if the extension's super-pattern is a sub-pattern of the to-be-extended pattern. This implies that the extension pattern is a sub-pattern of the to-be-extended pattern.

The prototype for `ColorWindow` is built incrementally by taking a copy of `Window`'s prototype and extending it. It is not the purpose of this paper to document prototypes; see [Mad93] for that. Only the interesting aspects of the prototype, especially those that differ between `Window` and `ColorWindow`, are elaborated.

`Window` has a do-part and a virtual declaration. Hence, its prototype will have an Inner Dispatch Table (IDT) and a Virtual Dispatch Table (VDT). This discussion will focus on these tables.

**The IDT.** This is the basis for the implementation of Beta's `INNER` imperative.

For a pattern `P` with do-parts, the number of entries in its IDT is one greater than the number of patterns in its super-pattern chain. In general, it has the following structure:

```
RETURN
P do-part
immediate-super-pattern do-part
...
base-pattern do-part
```

Here, `base-pattern do-part` is the do-part of the root pattern of `P`'s super-pattern hierarchy chain, `immediate-super-pattern do-part` is the do-part of `P`'s immediate super pattern, and `P do-part` is `P`'s own do-part. `RETURN` results in a return from subroutine.

In the case of `Window`, there is no super-pattern; hence only one do-part, and an IDT with two entries. In the case of `ColorWindow`, `Window` is a super-pattern; hence it has an IDT with three entries. Figure 4.2 shows the IDTs for both patterns. Observe how `ColorWindow`'s IDT is constructed out of parts of `Window`'s IDT. `Window`'s do-part is compiled into object-code and Entry 1 of both IDTs point to the entry-point for this object-code. Entry 2 of `Window`, and Entry 3 of `ColorWindow` point to code that executes a return-from-subroutine instruction. Entry 2 of `ColorWindow` points to "object-code" for the do-part of `ColorWindow`.

The object-code for the do-part of `ColorWindow` is interesting in that it is an interpreter-generated stub which serves the purpose of disguising interpreted code as compiled code. It is structured as follows:

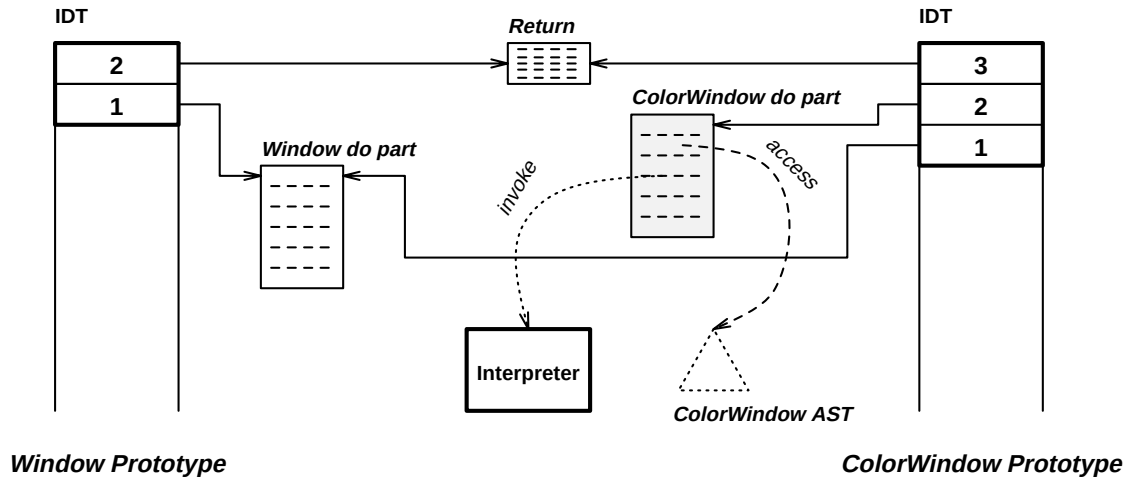


Figure 4.2: The Inner Dispatch Tables

1. Save registers
2. Get address of current object from a machine register
3. Locate AST for `ColorWindow`
4. Invoke interpreter with the current object and a reference to the AST
5. Restore registers
6. Return from subroutine

Notice that it has encoded within it a reference to the AST of `ColorWindow`.

When an instance of `Window` is executed, control will flow into Entry 1 of `Window`'s IDT. When the `INNER` imperative is encountered, the next entry (Entry 2) in the table will be invoked as a subroutine. As this is a return-from-subroutine instruction, the `INNER` will have the effect of a no-operation.

When an instance of `ColorWindow` is executed, control will flow into Entry 1 of `ColorWindow`'s IDT, which is object-code shared with `Window`'s IDT. At the `INNER` it will flow into Entry 2. This is the interpreter-generated stub; it will invoke the interpreter with the current object and the `ColorWindow`-AST as arguments. When during interpretation, the `INNER` imperative is encountered, the interpreter will treat it like a no-operation as there are no further specializations to branch to. When the interpreter finishes with the do-part of `ColorWindow`, it will return to the stub. Here the registers will be restored and execution will continue where it left off in Entry 1.

**The VDT.** A pattern, with one or more virtual patterns declared within it, will have a VDT. The VDT is the basis for the implementation of virtual patterns in Beta. The size of the VDT will be equal to the number of virtual patterns. The purpose of the VDT is to allow dynamic binding of virtual patterns — for a virtual pattern, the corresponding VDT entry will contain the current binding. A VDT entry is essentially a reference to the prototype of the pattern which is the current binding.

For example, the `Window` prototype will have a VDT with one entry, i.e. for the `refresh` virtual. This will contain a reference to the prototype for the `refresh` pattern declared within `Window`. All clients of `refresh` will be dispatched through its VDT-entry in order to get its current binding in the form of a prototype. This prototype will then be used to execute or instantiate an instance of `refresh`. If `refresh` is invoked on a `Window` instance, its current binding will be the `refresh` pattern declared within `Window`.

`ColorWindow` introduces a further binding of refresh i.e. it specializes refresh by defining a sub-pattern of it. In the prototype for `ColorWindow`, the VDT-entry for `refresh` is updated by the interpreter to refer to the prototype for the further binding of `refresh`. This prototype is also built by the interpreter in the same way as the prototype for `ColorWindow` is built. Invoking `refresh` on a `ColorWindow` instance will result in the execution of the further binding.

**Other prototype parts.** Other parts of `Window`'s prototype that get copied into `ColorWindow`'s prototype, possibly with modification, are the Static Objects Table, the Dynamic Reference Table, and some other information describing the size of the object etc. In this manner a prototype for `ColorWindow` is built from the prototype for `Window`.

### Getting the environment object

Once the prototype for `ColorWindow` has been built, it must be packaged with a reference to its environment object to form a pattern closure. Its environment object must be an instance of the pattern enclosing it — in the case of `ColorWindow`, this must be an instance of `Display`. But this object is also the environment object for `Window` — as `Window` and `ColorWindow` are processed in the same lexical scope. A reference to this object is thus obtained from the pattern closure for `Window`, which is available as an input parameter. The resulting pattern closure for `ColorWindow` may now be returned to the caller, where it may be used to instantiate or execute the extension pattern.

Figure 4.3 depicts the state after the extension pattern closure has been constructed. Notice the `ColorWindow`-closure which is returned by the interpreter.

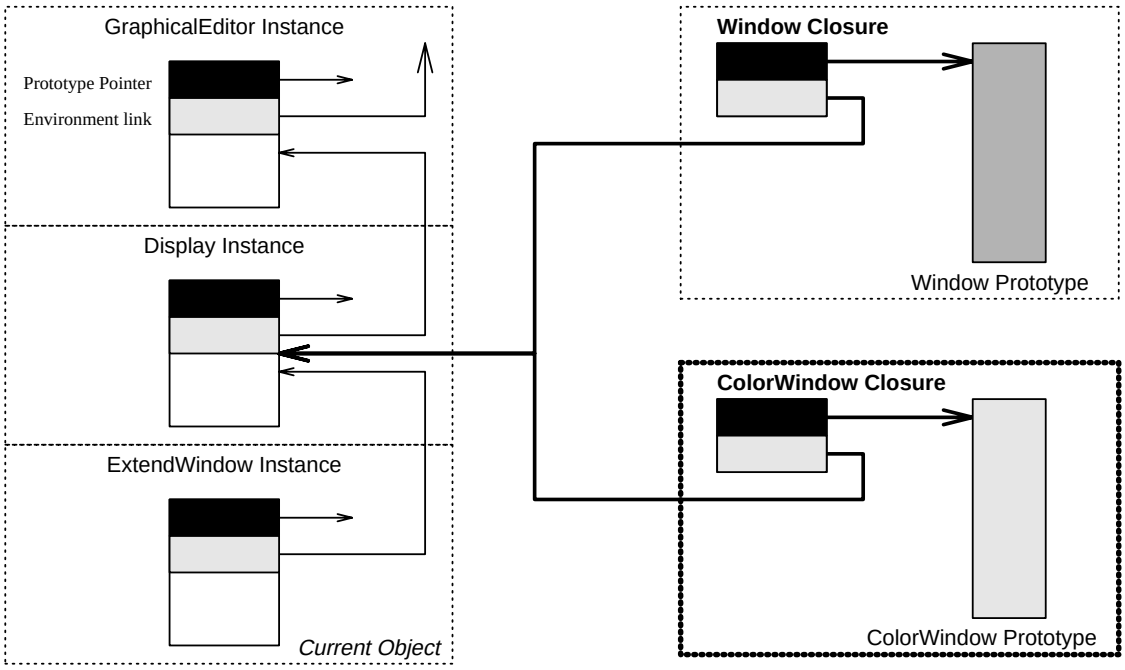


Figure 4.3: Illustrating the extension pattern closure

### 4.3.2 Invoking the returned pattern closure

Once the extension pattern is returned to the caller of the interpreter and stored in a pattern variable, it can be executed or instantiated. In `Display` the invocations of `WindowX` will result in instances of `ColorWindow` being created. The instance will be built using the prototype which is part of the pattern closure. The environment link for this new instance will be setup to point to the environment object which is stored in the pattern closure. Figure 4.4 shows an instance of `ColorWindow`.

Executing this instance of `ColorWindow` will result in the invocation of the do-part of `Window` (compiled). Upon execution of the `INNER` imperative, control will be transferred to the do-part of `ColorWindow` (compiled stub) from where control will flow into the interpreter. The interpreter will handle references to variables such as `screen` by traversing one environment link (to the `Display`) object and accessing the field at a pre-computed offset.

Calling `refresh` on this instance will result in a lookup of its VDT where a prototype for the further binding of `refresh` will be found. This will be used to create an instance of the extended `refresh` and execute it. Its execution will behave in a manner similar to that of `Window`: the do-part of the original `refresh` will be invoked, followed by the do-part of the further binding, when the `INNER` is executed.

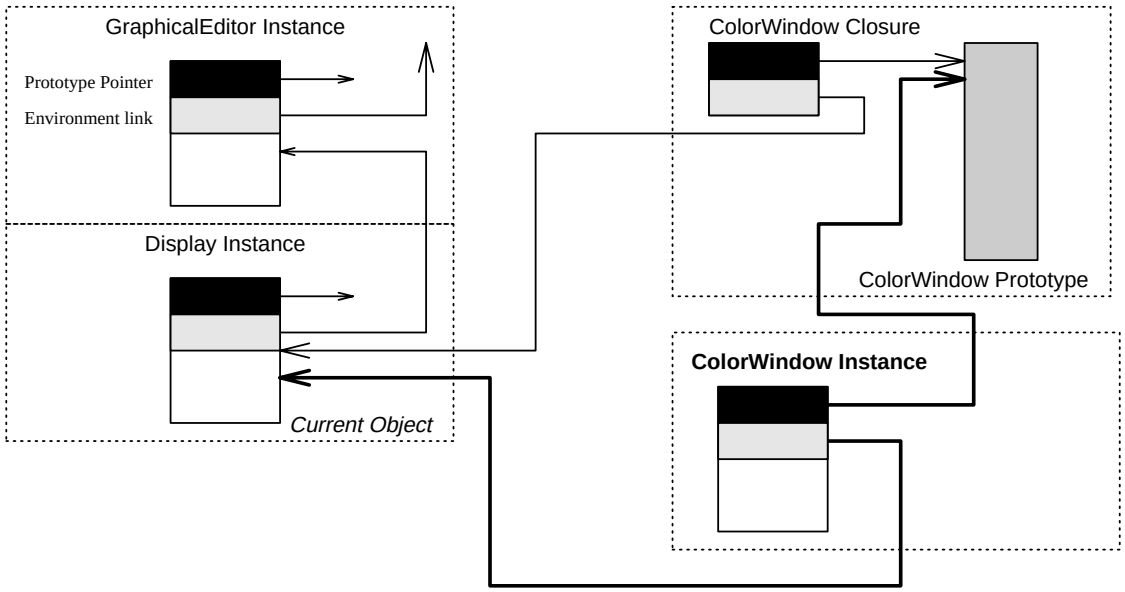


Figure 4.4: An instance of `ColorWindow`

## 4.4 Concluding Remarks

This paper has described the interesting aspects of the implementation of an interpreter for Beta. As shown, the interpreter may be easily embedded within any Beta application, thus making the application dynamically extensible. It has been shown how the implementation deals with issues such as dynamic extensions in a static language, allowing extensions access to the application's internals, interpretation contexts, intermixing of compiled and interpreted code, ensuring the type-safety of the application, packaging the extension in order to return it to the application.

An approach for building extensible applications that are able to extend themselves at well-defined points has also been suggested. These points get defined at application-development time and are designed to be able to accepts a range of extensions — any extension that is compatible with the specification of the extension point (i.e. is a sub-pattern of the type of the pattern variable). The extensions themselves are written in the original development language — in this case Beta — and are processed by the application by invoking an interpreter embedded within the application. The extensions may be placed, lexically, in any scope of the original application. This determines the parts of the original application that become visible to the extension.

This paper also documents, in a concrete manner, an attempt to integrate compiled and interpreted code in a manner traditionally present in Lisp systems, the difference being that here, it is done in the context of a statically-typed block-structured language with the possibility of integration at nested block levels.

The interpreter is a central element of this approach. Its ability to operate under these constraints makes its implementation interesting. The paper has concentrated on describing the interesting aspects of the implementation — it has not described every detail. The aspects that deal directly with its ability to process extensions have been presented. The ability to handle extensions in arbitrary scopes has been explained in detail. The construction of a prototype for the extension pattern, using the prototype for the pattern being extended, has been described. The basic idea behind this has been demonstrated by illustrating the construction of the IDT and VDT. The construction of pattern closures has also been demonstrated. All the necessary type-checks, and techniques to handle them, have been identified. All of this has been glued together by an illustration of the execution of an extension pattern.

There has been an emphasis on presenting the overall approach rather than describing a particular implementation — although at times details from a particular implementation have been used to make ideas more concrete. As a result, the techniques may be used for building interpreters for other languages, and more generally for handling extensions with or without interpretation. In fact, in the construction of the IDT entry for `ColorWindow`'s do-part (Section 4.3.1), the object-code which serves to interface with the interpreter could just as well have been replaced with object-code which implements the do-part directly. In this case, the core of the interpreter, not documented in this report, could be eliminated. All other ideas presented here would still be applicable.

The core of the interpreter is the part that does the actual interpretation — it traverses the AST, executing do-parts, processing imperatives, evaluating expressions, and accessing and modifying state. It is what gets invoked from the compiled do-part of `ColorWindow`. Its implementation has not been described in this paper as it is quite straightforward to implement. Our implementation interprets ASTs directly. It is written in an object-oriented style — e.g. an expression has a virtual `eval` method which is specialized in a multiplication expression, and so on. So, every syntactic category in the grammar of Beta is described by a Beta pattern; nodes of the AST are then instances of these patterns.

The material in this paper should be sufficient, when used in conjunction with the paper describing the Beta compiler and runtime [Mad93], and the source code of the interpreter, to gain a complete understanding of the interpreter's implementation. A missing element is the technique used to transfer control from the interpreter-generated object-code stubs to the interpreter with the current object and the correct AST. Hence, this paper can be used by someone maintaining or extending the interpreter.

## 4.5 Other Work

An alternative approach to building extensible systems in Beta was proposed in [AF89]. The primary differences between the two approaches can be summarized as follows: (1) they compile the extensions, and then dynamically load and link them; hence they don't require an interpreter (2) the invocation interface of their loader differs in minor ways from the interpreter. The use of a compiler and dynamic linker/loader implies more efficient extensions, but at the expense of slower turnaround time.

In their system, an extension is loaded in the following way:

```
(# dCar : ^CarDynStruct;
   (environment[], 'lada.ext') -> loader (# resultSP ::< Car #) -> dCar[];
   &dCar.p;              (* Execute a Lada *)
#)
```

Here `environment` is an object instance; the extension pattern is checked as if it was textually inserted at the place in the program where the attributes of the environment object are declared. This is similar to what happens in the interpreter approach; instead of an environment object, we provide the to-be-extended pattern closure. The two approaches are semantically equivalent, although the interpreter approach, in which a closure is passed as input, and an extended closure is returned as the result, is conceptually clearer.

The file `'lada.ext'` contains the extension pattern in source code form. The `loader` uses the Beta compiler to process the extension and then loads it into the calling program. The extension is compiled in a way so that external references can be handled at need-time. In other words, the dynamic linker doesn't resolve references immediately after the extension object-code is loaded; they get resolved if and when they are needed. This again is similar to what happens in the interpreter; references get resolved if and when they are used.

The return value of the loader, although similar in spirit to a pattern closure,[6] is an instance of a specialized `DynStruct` pattern. The details of this are beyond the scope of this paper. Pattern closures are a cleaner, and well understood, abstraction for all of these details.

Their report concentrates mainly on describing the implementation of the dynamic linker; the construction of prototypes for extensions is not described, as that is done by the compiler. This report, on the other hand, shows how prototypes for extensions can be incrementally constructed, in addition to showing how issues, similar in spirit to dynamic linking, are dealt with.

[6] Pattern closures were not a part of Beta when this system was built; the idea of pattern variables was proposed by them as a result of this work.

[Mal93a] describes the application programmer's interface of the interpreter described here. It also shows how the interpreter can be used for more than just extensible systems. [Mal94] presents general techniques for constructing extensible systems. It identifies language features which allow for the easy construction of extensible systems.

[Mad93] illustrates the implementation of the Beta compiler and runtime system. Prototypes are described in exhaustive detail in that paper. Building the interpreter didn't require any changes to the runtime system. The implementation techniques presented here could be used to implement the ideas contained in [KMMPN86]. [Lie87] describes an interpreter written in an object-oriented style.

Other environments which allow mixing of compiled and interpreted code include the Standard ML of New Jersey system [AM87], and various lisp and scheme implementations. GNU Emacs [Sta84] also has an embedded lisp interpreter which allows for free intermixing of compiled and interpreted code.

The techniques presented here allow an application to be treated as a black-box (abstraction), but with well defined open points that expose its implementation and allow it to be changed. A more general, and highly motivating, discussion on building abstractions that are open is presented in [Kic92]. Suggested here, is the idea that the traditional notion of abstraction is not very useful for many application-building efforts, and that a slightly modified notion of abstraction, in which one has the traditional interface (for using the abstraction) and a meta-level adjustment interface, is necessary. [KdRB91] is an example of a system with such a dual interface. This work accomplishes some of the goals outlined in [Kic92].

## Acknowledgements

This work has been generously supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. Thanks to Ole Lehrmann Madsen for his guidance during the implementation of the interpreter and to Peter Andersen for helping with the runtime system. Thanks to Kim Jensen Møller for his assistance with the meta-programming system.

# Chapter 5

# Extensibility as the basis for Incremental Application Generation

**Author:** Jawahar Malhotra

**Date:** November 1993

**Publication Information:** Aarhus University Technical Report [Mal93b]. To be submitted as a conference paper.

**Abstract.**

This paper describes an approach to making a direct-manipulation-based user-interface generator incremental in its ability to generate and execute new applications. The approach is best applicable for user-interface generators which generate code in high-level languages without incremental compilers. By adopting this approach, a user-interface generator can reduce, significantly, the length of time between an edit of some user-interface component and the generation of a new executable incorporating the edited component. The approach relies heavily on the notion of extensibility, as found in object-oriented languages, and uses a technique called incremental code-generation. The approach is demonstrated by means of an example of a dialog taken from the Macintosh Finder. It is shown how various edit operations can be handled incrementally. The presentation is in the context of a user-interface generator called the ApplBuilder which generates code in the object-oriented language Beta. A section is devoted to illustrating how the ApplBuilder can be adapted to adopt this approach.

## 5.1 Introduction

Direct-manipulation based user-interface generators are used to build complex user-interfaces in a fraction of the time it would take to accomplish the same task manually. These tools allow their users to edit a user-interface using direct-manipulation techniques; they then generate code describing the user-interface component in some target language — usually a high-level language like C, C++, CLOS, Beta. The remainder of the user's application can utilize this user-interface component by simply linking with this code — and of course invoking it. If a user-interface component needs to be changed, the user edits it using the tool and regenerates its code.[1] This code must then be recompiled and relinked with the remainder of the application to produce an application with the edited UI component. This process may also trigger recompilation of other parts of the user's application. If the target language has an incremental compiler, this process of regeneration of the new executable is relatively quick. If, on the other hand, all the language has is a batch compiler, this process can be unacceptably lengthy. This reduces the effectiveness of the tool in general, and renders the tool useless for purposes such as rapid prototyping.

This paper proposes an approach which helps reduce the length of this regeneration process. The approach uses the notion of *extensibility* [Mal93a, Mal94] along with a technique called *incremental code-generation* to produce results that are comparable with incremental compilation. Incremental code-generation is explained in the sequel.

The benefits of reducing the length of the regeneration process are manifold. First and foremost the user-interface generator feels more interactive; as a result, user productivity is greatly enhanced. In addition, users are encouraged to experiment, as the results of their experiments are quickly observable and easily correctable if necessary. Furthermore, the generator can be used as a rapid-prototyping tool.

Another possible approach to reducing this regeneration time is based on incremental compilation [Hed92, HM86]. With an incremental compiler available for the target language, the regeneration process can be made entirely interactive: each edit command issued by the user is translated by the user-interface generator into a transformation of the underlying source-code, which is processed by the incremental compiler, which recompiles the necessary parts and patches the existing executable to include the changes. On the other hand, if only a traditional batch compiler is available, the user-interface generator is forced to regenerate code for the affected module(s); it is also necessary to

[1] The user could also edit the source-code directly; it is assumed that the user *doesn't* do this. This assumption is entirely justifiable in systems where interfacing with the automatically-generated code doesn't require any in-place modification of it. Hence, regenerating code will not invalidate any of the user's work.

recompile these module(s) (and all others that depend on it), and to relink the entire application.

The approach proposed here lies somewhere in the middle of the continuum between batch compilation and incremental compilation. It is designed specifically to give the effect of incremental compilation without being as general, and hence as complicated, as incremental compilation. With this approach, user-interface generators with non-incrementally-compiled target languages can be made interactive.

Figure 5.1 illustrates the approach. There are two possible ways to go after an edit operation: the traditional *batch* route, or the new *incremental* route. Along the batch route, new code is generated for the edited user-interface (UI) component. Its old code is *replaced* by its new, and the affected modules recompiled. The application is then relinked. Along the incremental route too, new code is generated for the edited UI component. This code, however, is generated as an *extension* of the old code. No parts of the original application's code are modified. This form of code generation, where extensions are generated, can be thought of as *incremental code-generation*.

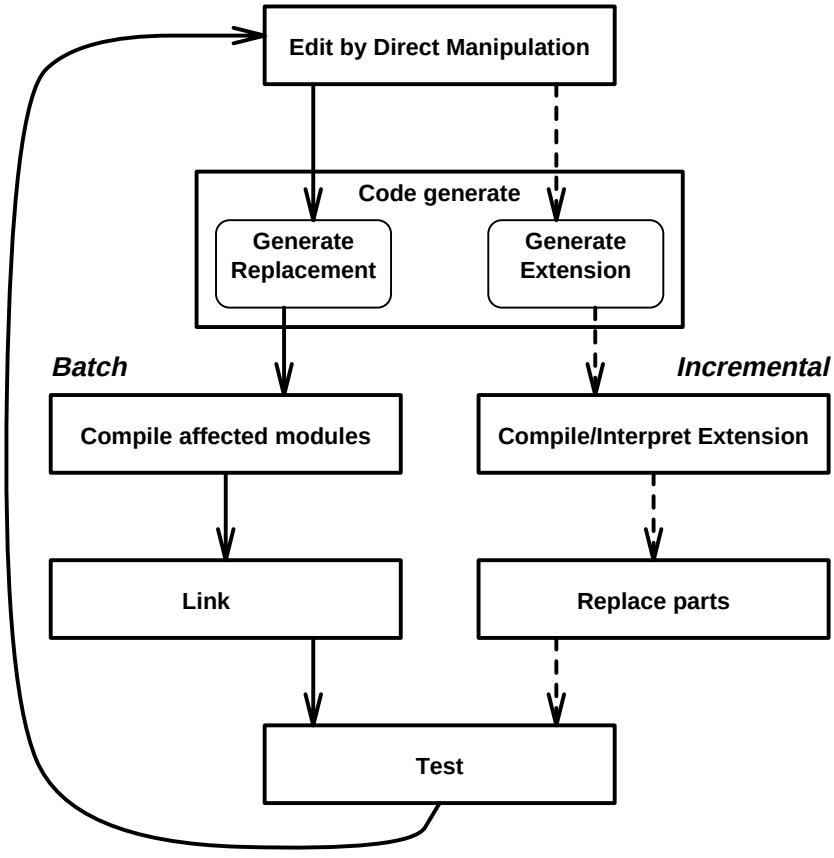


Figure 5.1: The Incremental Extension approach

Intuitively, these extensions are "replacement parts" for the original application. These

extensions are such that they must be compiled or interpreted in the context of the original system, for they may refer to parts of the system. There is no need, however, to recompile any parts of the original system; even parts that refer to the replacements don't need recompilation. A new executable is obtained by replacing parts of the original executable with their replacements. The new executable is equivalent in functionality to the executable one would have obtained, had one followed the batch route.

The advantage over the batch approach is that it is necessary to compile or interpret only the extensions, not all parts that depend on it. In the batch approach, it becomes necessary to recompile modules containing the modifications as well as all other modules that depend on them. Hence, the proposed approach requires less work. With a compiler or interpreter of fine granularity (e.g. function/procedure/class level) this work can be further minimized. Furthermore, instead of linking the recompiled modules (as in the batch approach), the proposed approach calls for a replacement of the original part by the replacement part — a procedure simpler and quicker than general-purpose linking.

This approach is also simpler than general-purpose incremental compilation. The main complexity of incremental compilation is the dependency analysis. This approach doesn't need any dependency analysis due to the fact that extensions don't require recompilation of their dependents. What is shared in common with incremental compilation is the need to replace parts of the original executable by their newly-compiled replacements.

The approach has been tested in the context of an application generator called the ApplBuilder [GHT91][2]. The ApplBuilder generates code in the object-oriented language Beta [MMPN93]. The code generated by the ApplBuilder is organized so that user-written code can interface to it without any in-place modifications of it. Thus, regenerating code doesn't invalidate any user-written code. The ApplBuilder is also unique in supporting the reloading, and subsequent editing, of a user-interface component whose code has been edited textually.

Beta is a statically-compiled object-oriented language in the tradition of Simula [DMN68]. Among other things, it has the notion of a pattern as the one and only abstraction mechanism. The Beta pattern unifies classes, procedures, functions, types, and various other abstraction mechanisms found in other languages. Beta patterns may be nested to arbitrary levels. Beta doesn't have an incremental compiler and hence the ApplBuilder exhibits the symptoms described earlier. The ApplBuilder is able to minimize the amount of code generated in response to an edit operation. In fact, for many types of edit operations, the ApplBuilder is able to restrict modifications and recompilations to just a single module. For other types of edit operations, it restricts modifications to a single

[2]An application generator is a generalization of a user-interface generator in that it supports the construction of the entire application and not just the user-interface.

module, but as a result of that modification, dependent modules are also recompiled. In any case, the application must be relinked with the recompiled modules before it can be tested. The ApplBuilder has therefore, been the motivation, and a good example, for the approach presented here.

The approach is presented in the context of the ApplBuilder and Beta. It relies, however, on the notion of extensibility found in most object-oriented languages. Hence, it is applicable to most object-oriented languages. The semantics of Beta differ from most other object-oriented languages, especially in the area of refinement versus overriding. Beta methods refine their super-methods, with the `INNER` construct being used to transfer control to the refinement. In most other languages, methods override their super-methods, requiring an explicit call to `super` to invoke the super method. The approach illustrated here is based on Beta's semantics, but could be easily generalized to other languages.

The approach also relies on the availability of an interpreter or dynamic linker. Only recently has Beta become an interpreted language [Mal93a, Mal94].

The paper demonstrates the incremental code-generation approach by means of a real example: the Find File dialog from the Macintosh Finder (Section 5.2.1). It is shown how edits of this dialog are handled by this approach (Sections 5.2.2 and 5.2.3). It then shows how these edits can be composed (Section 5.2.4). Section 5.2.5 shows how the incrementally-generated extensions can replace their original counterparts without any recompilation of the original application. The ApplBuilder must be adapted to use this approach; this is the subject of Section 5.3. The paper concludes with more comparisons between incremental compilation and this approach, along with a mention of other benefits of this approach.

Minimal knowledge of Beta is assumed. Readers wishing to acquaint themselves with Beta can find a short primer in [Mal93a]; the book [MMPN93] provides a more systematic and comprehensive introduction to the language.

## 5.2 The Approach

Suppose one has a program (*base program*), with some user-interface component (*base UI component*), generated by the ApplBuilder. Now, suppose one edits the base UI component via the ApplBuilder. One gets the new program with the edited UI component using the following approach, termed the *incremental approach*:

1. Express the edited UI component as an extension of the base UI component. More specifically, the interface of the edited UI component should be an extension of the

interface of the base UI component. It should be possible to safely replace the base UI component by the edited UI component without recompiling any of the clients of the base UI component.

In practice this implies that the type of the edited UI component should be a sub-type of the type of the base UI component. The class describing the edited UI component must be a sub-class of the class describing the base UI component. This is accomplished by generating the source-code for the edited UI component such that its super-class is the base UI component. This technique is employed even when the edited UI component is, *functionally*, *not* an extension (e.g. has one less button) of the base UI component.

2. Replace the base UI component by the edited UI component in the compiled, or even executing, base program. Due to (1), this can be done without any need to recompile the clients of the base UI component. In fact, this is accomplished without any recompilation of the base program.

The new program, thus obtained, is functionally equivalent to the program one would have obtained using the *traditional approach*: i.e. generate code for edited UI component without ensuring that it is an extension of base UI component, replace — in the base program — the source-code for base UI component by source-code for the edited UI component, and recompile the base program.

The incremental approach for building the new program is clearly more efficient than the traditional approach, the primary reason being that, in the incremental approach, the only compilation required is that of the edited UI component. The traditional approach, however, requires the recompilation of, at least, the edited UI component along with all its clients. In addition, if the separate compilation system of the language is not fine-grained enough, it may force the recompilation of many additional parts: the entire modules containing the clients will get recompiled.

For the incremental approach to work, it must be shown that if $C_1$ is a UI component obtained from $C_0$ by a sequence of well-defined edit operations, then $C_1$ can be expressed as an extension of $C_0$. The interesting edit operations are (1) addition of a new sub-component, e.g. a button, (2) removal of a sub-component, e.g. removal of a button, (3) modification of a sub-component e.g. modification of the action script of a button. These should cover all the operations allowable by the ApplBuilder. In fact, (3) can be accomplished by doing (2) to remove the component, and (1) to add the modified component. So, this report will concentrate on (1) and (2).

This is proved by means of a real example in which a dialog is edited, first to add a new button to it, then to remove a button from it. In each case, the source-code for the

edited dialog, generated as a sub-class of the base dialog, is shown, and illustrated to be functionally correct. Although all the code for this example is presented here in abstract form, it is real code that has been tested and has proved to work.

The section proceeds by first illustrating the example dialog and the source-code generated by the ApplBuilder for it. The next two sub-sections show how edits (addition/deletion) of the dialog are handled. That this approach works even when edits are composed, is illustrated in the following section. Finally, it is shown that these extensions, representing the edited UI components, can be made to replace their corresponding base UI components without any need to recompile the base program.

### 5.2.1 The original Find File dialog

Consider the dialog presented in Figure 5.2.[3] This dialog has been built using the ApplBuilder. The code generated for this dialog is shown in Listing 1.[4]

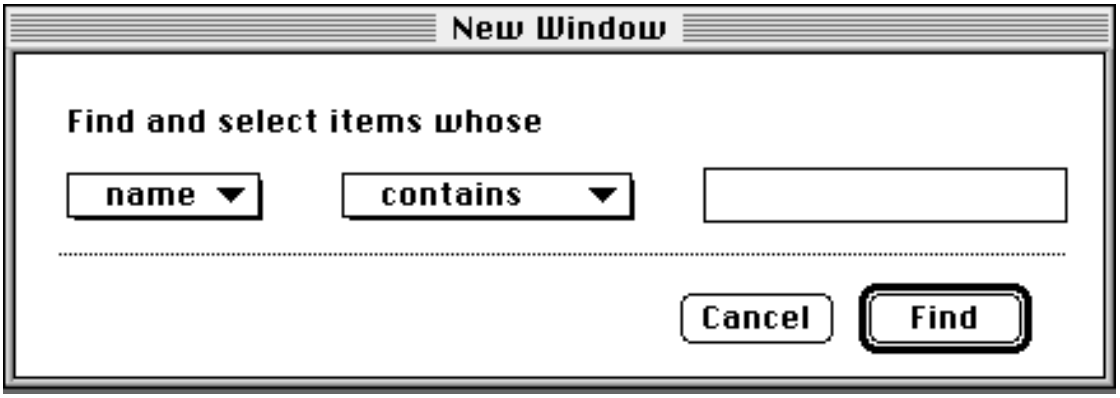


Figure 5.2: The *Find File* dialog

A pattern describing the dialog, called `FindDialog`, is shown nested within `mainProgram`. It is declared as a sub-pattern of `Window`. Within it are the declarations of the various components of the Find File dialog: `PromptLabel`, `ContainsMenuButton`, `NameMenuButton`, `SearchTextFld`, `CancelBtn`, `FindButton`, etc. Each of these components has an (inherited) `Open` method which is used to activate[5] the component. Some components, like `ContainsMenuBtn` and `NameMenuBtn`, further bind the `Open` method. For example, the further binding of `Open` in `ContainsMenuBtn`, in addition to activating the menu button, also activates the appropriate menu and associates it with the menu button.

[3]This dialog is similar in design to the Find File dialog in the Macintosh Finder

[4]Not exactly; the code shown here is slightly simplified for ease of presentation: in particular, it is not fragmented. For the purpose of this discussion, it is functionally equivalent to the code generated by the ApplBuilder.

[5]display on screen.

**Listing 1**. The generated code for the *Find File* dialog.

```
mainProgram: macenv
(#  findDialog: Window
    (#  PromptLabel: @StaticText;
        ContainsMenuBtn: @MenuButton
        (#  Open::< (# ... #);
        #);
        NameMenuBtn: @MenuButton
        (#  Open::< (# ... #);
        #);
        SearchTextFld: @EditText;
        Line1: @Rectangle;  (*  horizontal separator *)
        CancelBtn: @PushButton
        (#  mouseDown::<
            (# do this(Window).close #)
        #);
        FindBtn: @PushButton
        (#  Open::<  (# ... #);
            mouseDown::<
            (# do 'Finding ' -> putText;
                  searchTextFld.getText -> putLine;
            #);
        #);
        Open::<
        (#  do 1 -> PromptLabel.Open;
               2 -> ContainsMenuBtn.Open;
               3 -> NameMenuBtn.Open;
               4 -> SearchTextFld.Open;
               5 -> Line1.Open;
               6 -> CancelBtn.Open;
               7 -> FindBtn.Open;
               INNER;
        #);
    #);  (* findDialog *)
    theFindDialog: ^findDialog;
   ...               (* other patterns of mainProgram *)
do
   ... &findDialog[] -> theFindDialog[];
       1000 -> theFindDialog.Open; ...
#)
```

In addition to generating the Beta code describing the behavior of the component, the ApplBuilder also generates a resource which describes the visual attributes (size, geometry, etc.) of the component. This resource is stored as a Macintosh resource, and has a unique ID. This resource ID, supplied to the `Open` method as an argument, allows it to determine the visual attributes of the component it must display.

`CancelBtn` and `FindBtn` further bind a method called `mouseDown`. It is here that the action of the button is described.

In the do-part of `mainProgram`, the variable `theFindDialog` stores an instance of the pattern `findDialog`. This instance's `Open` is then called with the resource ID of the window for the dialog. When this happens, an instance of the window is displayed on the screen. Subsequently, the further binding of `findDialog`'s `Open` (shown in the listing) is invoked; this calls `Open` for each of the components of the dialog, causing them to activate themselves.

The reader should be convinced that the code in Listing 1, although not presented in complete detail, correctly describes the dialog in Figure 5.2.

### 5.2.2 Adding new components

Suppose the user of the ApplBuilder edits the dialog in Figure 5.2 to produce the dialog in Figure 5.3. The user has essentially added a new menu-button, called Search, to the original dialog. This section will show how the edited dialog can be represented by a Beta pattern that is a sub-pattern of the original dialog (Listing 1).

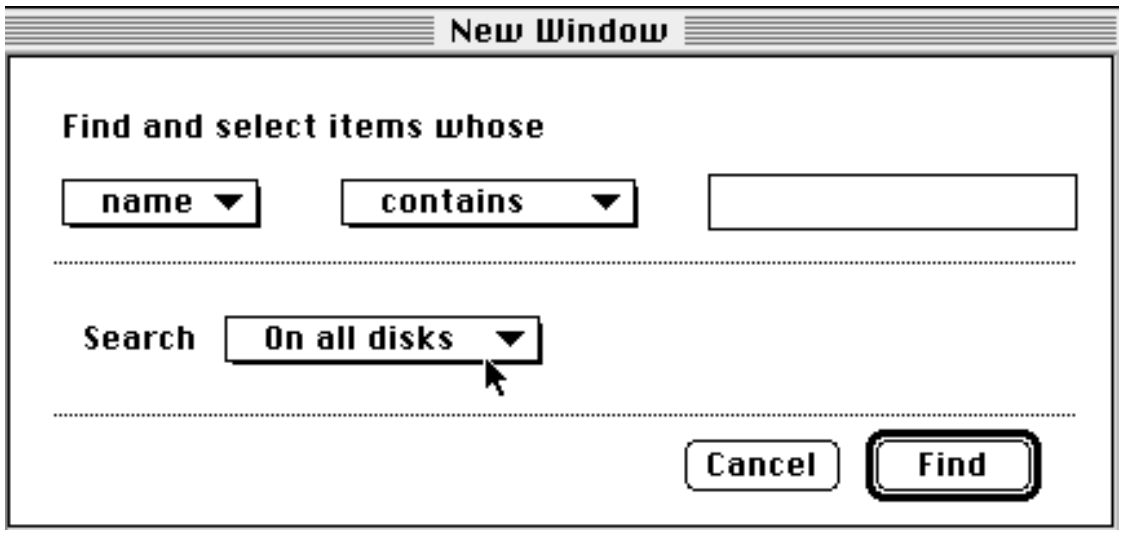


Figure 5.3: The extended *Find File* dialog

The code for the edited dialog is shown in Listing 2, ignoring `mainProgram` for now.

**Listing 2**. Adding the Search menu-button.

```
findDialogExt: findDialog
(#   SearchMenuBtn: @MenuButton
     (#  Open::< (# #); #);
     Line2: @Rectangle;
     Open ::<
     (# do 8 -> SearchMenuBtn.Open;
           9 -> Line2.Open;
           INNER;
     #);
#)
```

The pattern describing the edited dialog is called `findDialogExt`; it is declared as a sub-pattern of `findDialog`. Hence, it is clearly an extension of `findDialog` and can safely replace `findDialog` in the base program.

Functionally `findDialogExt` behaves exactly as expected (Figure 5.3). To see this, observe what will happen when its `Open` method is invoked. It will first invoke the `Open` method of `findDialog`; when the `INNER` imperative is reached, control will be transferred to the `Open` declared in `findDialogExt`. Here, an instance of `SearchMenuBtn` will be activated, resulting in the display of the Search button.

### 5.2.3 Removing components

Consider what would happen if the Contains menu-button, illustrated in Figure 5.2, must be removed. In this case, the edited dialog doesn't feel, at least *functionally*, like an extension of the original dialog: it has one *less* button. It is still possible, however, to generate a Beta pattern for the edited dialog so that it is a sub-pattern of the `findDialog`.

This requires that `findDialog` be structured a little differently to begin with. The desired structure for `findDialog` is shown next, and it is assumed that the original dialog is generated in this manner. This code is functionally equivalent to the `findDialog` in Listing 1. Generating such code should require a simple modification to the ApplBuilder. Note also that generating code in this way doesn't, in any way, invalidate the approach of the previous section: it is still possible to add new components; this requires extending the enabling vector (yet to be explained), an operation that is possible in Beta.

The code is shown in Listing 3.

**Listing 3**. An extensible version of the code for the original *Find File* dialog.

```
findDialog: Window
(#  PromptLabel: @StaticText;
    ... <<<identical to corresponding parts in LISTING 1>>> ...
    enabVector: [7]##Object;
    setupEnabVector:<
    (#  (* define patterns to enable each of the components *)
        openPromptLabel : (# do 1 -> PromptLabel.Open #);
        openContainsMenuBtn : (# do 2 -> ContainsMenuBtn.Open #);
        openNameMenuBtn : (# do 3 -> NameMenuBtn.Open #);
        openSearchTextFld : (# do 4 -> SearchTextFld.Open #);
        openLine1 : (# do 5 -> Line1.Open #);
        openCancelBtn : (# do 6 -> CancelBtn.Open #);
        openFindBtn : (# do 7 -> FindBtn.Open #);
     do
        (* store these patterns in the enabling vector *)
        openPromptLabel## -> enabVector[1]##;
        openContainsMenuBtn## -> enabVector[2]##;
        openNameMenuBtn## -> enabVector[3]##;
        openSearchTextFld## -> enabVector[4]##;
        openLine1## -> enabVector[5]##;
        openCancelBtn## -> enabVector[6]##;
        openFindBtn## -> enabVector[7]##;
        INNER;
    #);
    Open::<
    (# do setupEnabVector;
          (for i: enabVector.range repeat
             (* open each component *)
             enabVector[i];
           for);
          INNER;
    #);
#);  (* findDialog *)
```

With the exception of the definition of `enabVector` (enabling vector), `setupEnabVector`, and a different definition of `Open`, it is identical to the one in the original listing. `Open` uses the `enabVector` to activate the various sub-components (call their `Open` methods), rather than executing their `Open` methods directly. The size of the `enabVector` is equal to

the number of sub-components in the dialog, in this case 7. `setupEnabVector` is invoked before the `enabVector` is used; as it is virtual, it can be further bound in sub-patterns and used to disable one or more of the sub-components. Hence the name *enabling vector*.

Given that the original dialog is as in Listing 3, it becomes possible to generate code for the edited dialog as a sub-pattern of `findDialog`. This is illustrated in Listing 4.

**Listing 4**. Removing the Contains menu-button.

```
findDialogExt: findDialog
(#  setupEnabVector::<
    (# do Object## -> enabVector[2]##;
          INNER;
     #);
#)
```

Here the `setupEnabVector` method is further bound and used to assign the generic no-op `Object` into slot 2 of the enabling vector. As a result, when the `Open` method of this pattern executes the object in slot 2 of the enabling vector it will end up *executing a no-op*, instead of executing an object (`openContainsMenuBtn`) which would have called the `Open` method of `ContainsMenuBtn`. The net effect of this will be that the resulting dialog will not have a Contains button.

### 5.2.4 Composing edits

The previous sections have established that it is possible to generate the edited dialog as a sub-pattern of the original dialog. This section will establish that if an edited dialog, whose code has been generated as an extension, is edited further, the resulting dialog can still be generated as an extension.

Consider the example of the previous sections: the first edit adds a Search button, while the second removes the Contains button. The first edit results in code for `findDialogExt` defined as a sub-pattern of `findDialog`. The second edit can be handled by generating its code as a sub-pattern of `findDialogExt`. Assuming that the original code was generated with the enabling vector, this code may look as follows:

**Listing 5**. Composing edits.

```
findDialogExt2: findDialogExt
(#  setupEnabVector::<
```

```
    (# do Object## -> enabVector[2]##;
          INNER;
     #);
#)
```

Clearly, `findDialogExt2` incorporates both the edits and is an extension of `findDialogExt` and hence of `findDialog`.

An alternative is to discard the code generated after the first edit. In this case, the pattern generated is a direct sub-pattern of the base UI component `findDialog` and incorporates all the edits, first and second in this case, to the original dialog.

Either approach will work equally well. Incorporating all the edits into a single extension will prevent the proliferation of many extensions. On the other hand, having many small extensions will restrict compilation to smaller units. In any case, it should be clear that the approach works even when edits are composed.

### 5.2.5 On-the-fly incorporation of extensions

The previous sections have shown how edits, possibly composed, of a UI component can be handled by generating code which is an extension of the original code. This section will show how these extensions can be incorporated into the base program without any need to recompile the base program. This will complete the explanation of how one can obtain a new program with the edited UI component(s).

Consider the extension `findDialogExt` presented in Listing 2. This section will show how `findDialog` declared in the base program (Listing 1) can be replaced by this extension without any recompilation of the base program. The technique is based on preparing the base program to handle such replacements of components by their extensions. The technique calls for a different code-generation style for the base program.

Listing 6 shows the base program generated in a different style. This code is functionally equivalent to the code in Listing 1; the differences lie in the coding style. This version is more open for extensions; hence it will be called the *extension-ready* version. Generating such code, instead of the original code, should require a simple modification to the Appl-Builder. Note, once again, that generating code in this way doesn't, in any way, invalidate the approach of the previous sections: the `findDialog` pattern is unchanged and hence its extensions can be generated as explained earlier.

**Listing 6**. The extension-ready code for the *Find File* dialog.

```
mainProgram: macenv
(#  findDialog: Window
    (#  ...
           <<<identical to corresponding descriptor in LISTING 4>>>
        ...
    #);  (* findDialog *)
    extend:<
    (# do findDialog## -> extensionServer.getPattern -> findDialogP##;
          1000 -> extensionServer.getResourceId -> findDialogRsrc;
    #);
    findDialogP: ##findDialog;
    findDialogRsrc: @integer;
    theFindDialog: ^findDialog;
do
   extend;
   &findDialogP[] -> theFindDialog[];
   findDialogRsrc -> theFindDialog.open;
#)
```

The primary difference lies in the way in which the `findDialog` pattern is used: it is accessed via a pattern-variable called `findDialogP`. The `extend` method is capable of assigning a new pattern to `findDialogP` and is executed before `findDialogP` is used. It queries an extension server (using `getPattern`) for an extension of the `findDialog` pattern, storing the resulting pattern in `findDialogP`. Similarly, it also gets the resource ID of the extended pattern. Assuming that the extension server is the identity function (i.e. both `getPattern` and `getResourceId` return their arguments), this code will behave identical to the code in Listing 1 and will create the original dialog shown in Figure 5.2.

The `getPattern` method of the extension server is a map from pattern to pattern: given a pattern, if an extension for the pattern has been registered with the extension server, it will return the extension pattern, else it will simply return its input. Similarly, `getResourceId` will return the resource ID corresponding to the extension pattern, if any; it will return its input if an extension has not been registered.

A program which uses the extension server, as does `mainProgram`, can be made to use extended patterns by simply registering the extended patterns with the extension server. The extension server can be implemented as a database external to the program, thus making the extension-registering process independent of the program execution. It also serves to make the extensions persistent.

In the Beta system, every pattern has a unique ID. The extension server's `getPattern` method can be implemented as a map from such a pattern ID to the object-code of the extension. In order to install an extension for a pattern, e.g. `findDialog`, one can define an entry mapping the ID of the `findDialog` pattern to the object-code of its extension pattern. This object-code is obtained by compiling the extension pattern in the right context. Once the extension is registered, when the program executes the statement:

```
findDialog## -> extensionServer.getPattern -> findDialogP##;
```

`getPattern` is invoked with the ID of the `findDialog` pattern. It can use this ID to search the extension-map for a matching entry. If found, it can take the associated object-code and dynamically link it into the calling program; the code can then be packaged into a structure value denoting the extension and returned by `getPattern`. The report [AF89] shows that such an approach, where extensions are compiled and dynamically linked, is entirely feasible.

Alternatively, the extension server could map pattern IDs to the source-code of the extension. In that case, `getPattern` could interpret the source-code of the extension, packaging it into a structure value to be returned. This approach is described in detail in [Mal93a, Mal93c].

The `getResourceId` method of the extension server is trivial to implement: it simply maps integers to integers. When an extension is registered for a pattern, an entry mapping the pattern's resource ID to the extension's resource ID must also be created.

So, to replace `findDialog` by `findDialogExt` in `mainProgram`, one would compile `findDialogExt`, and register it with the extension server, using the ID of `findDialog` as the key. When `mainProgram` executes `getPattern` it will find this entry and assign `findDialogExt` into `findDialogP`. Subsequently, all uses of `findDialogP` will denote `findDialogExt`.

### 5.2.6 Summary

This section has shown the following:

1. If the base program has been generated as in Listing 6
2. and the user-interface components have been generated as in Listing 3
3. then edits of the user-interface components can be handled by
4. generating the source-code of the new user-interface component as an extension of the original component's source-code

5. and replacing the original by the extension without any recompilation of the base program.

## 5.3 Adapting the ApplBuilder

The previous section has illustrated an approach that can be employed in order to achieve incremental application generation and execution. This section will show how the ApplBuilder should be adapted to use this approach.

Currently, the ApplBuilder is able to generate code in the following ways: (1) when a new application is created, it generates code for the entire application, and (2) when an existing application is edited, it regenerates code for only the affected modules. The code produced in (1) is *not* extension-ready; it is not prepared to accept extensions as in Listing 6, and its user-interface components are not extensible as in Listing 3. The code produced in (2) is produced as a replacement of the old code, and not as an extension.

The ApplBuilder must therefore be extended to have two other modes of code generation:

1. *Extension-ready code-generation.* In this mode, code for the entire application is generated, but in an extension-ready manner. An application generated as in Listing 6, with its user-interface components generated as in Listing 3, is an example of extension-ready code-generation.

2. *Extension code-generation.* In this mode, the code for the edited UI component is generated incrementally as an extension of the original UI component's code. Listing 2 and 4 illustrate this style of code generation. Clearly, this style of code generation can work only if the base program has been generated in an extension-ready style.

The ApplBuilder could provide these code-generation styles as options to its users. During the prototyping and system-development phase, the user could generate the base program using the extension-ready style. This would allow subsequent edits to be generated as extensions, and hence tested without lengthy compilation and linking phases. For the final version, the traditional approach could be employed, producing code that is more efficient. At the same time, it should be noted that since the execution overheads of user-interfaces are high — mostly due to the intensive IO requirements — inefficiencies introduced by these new code-generation techniques may be insignificant. In that case, it may be better to use these code-generation techniques in all phases.

In order to generate code that is extension-ready, one must introduce pattern-variables and calls to the `extend` pattern (Listing 6). Which patterns should be accessed via pattern-variables, or where should the `extend` patterns be placed? In other words, which patterns must be *replaceable*?

An approach that could work is to make all *composite* user-interface components replaceable, and to make all others non-replaceable. By composite is meant a component that is defined as a composition of other components. So, for example, `findDialog` is a composite as it is composed of many buttons and other components, whereas `ContainsMenuBtn` is not a composite, as it a primitive component.

In order to generate code incrementally, the following algorithm can be employed:

1. If the edited UI component is replaceable (e.g. `findDialog`), generate code for the edited UI component.

2. If the edited UI component is not replaceable (e.g. if `ContainsMenuBtn` is edited) then generate code for the smallest enclosing replaceable UI component. This implies that even a small change in the script of the `ContainsMenuBtn` will require generating an extension to `findDialog` as `ContainsMenuBtn` is not replaceable in `findDialog`. But, this overhead is not expected to be very high, and more than compensated by the reduction in the number of pattern-variables and `extend` patterns attained as a result of making only composite components replaceable.

In addition to having these new code-generation techniques, the ApplBuilder should also be able to register extensions with the extension server (see Listing 6). This should pose no major problems. Figure 5.4 highlights the primary architectural difference between the original ApplBuilder and the proposed ApplBuilder.

## 5.4 Conclusions

An approach which allows a direct-manipulation based interface builder, such as ApplBuilder, to generate code incrementally, in response to edit operations, has been described. The approach is contingent on generating the application in an extension-ready manner. Edited versions of components of the original application are generated as extensions of their original counterparts. The originals are then replaced by their corresponding extensions without any need to recompile the original program. One possible way to adapt the ApplBuilder to use this approach has been demonstrated.

Such an incremental code-generation approach is clearly more efficient than the traditional non-incremental approach. It is clear that it avoids much unnecessary recompilation.

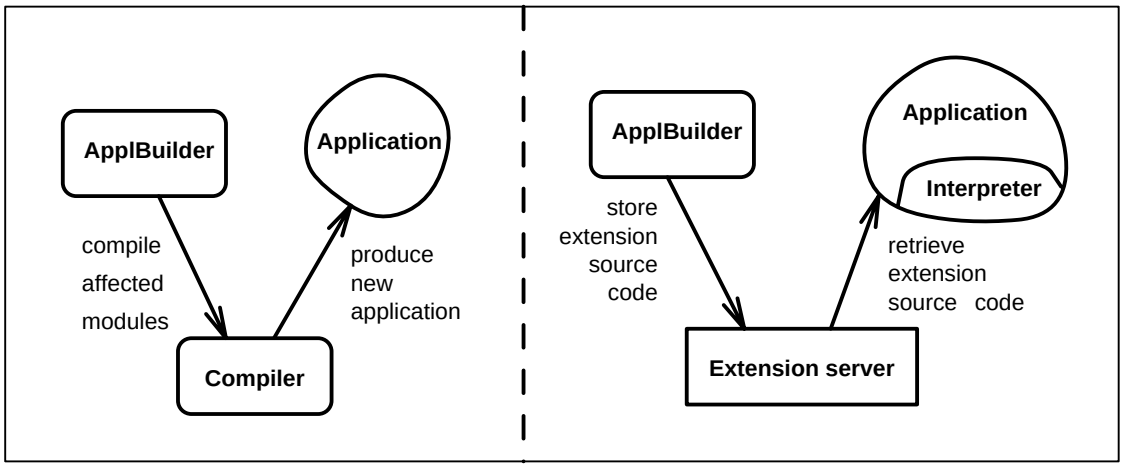


Figure 5.4: ApplBuilder architectures

In the examples, the traditional approach would have required a recompilation of the newly generated `findDialog` along with all its clients. This is because there would be no guarantee that the new `findDialog` satisfies the interface of the old one. The incremental approach also avoids the lengthy link process which is a severe bottleneck in the current system. The code-generation constraints imposed on the base program in order to make it extension-ready clearly add some overhead. This, however, is not a problem in this context as the I/O overhead involved in creating user-interface components more than shadows this overhead.

Incremental compilation, on the other hand, is a more general approach. With it, the ApplBuilder would be freed of most concerns of incrementality. There would be no need to impose artificial code-generation constraints. In response to an edit of some part of the application, the ApplBuilder could generate the new code for just that part. There would be no need for this new code to be an extension of the code it was replacing. The incremental compiler could be given this code along with an indication of what, in the original program, it was replacing. The incremental compiler would produce the new executable, ensuring that all clients that need recompilation are recompiled, and, at the same time, minimizing the amount of compilation necessary.

The approach presented here has the potential of being more efficient than the incremental compilation approach. This is primarily because the incremental compilation approach will do more work: it will do some analysis to figure out the clients of the code being replaced/edited and, it will recompile these clients. Both of these are avoided by the approach presented here. The reason why this is possible is that the approach presented here imposes some structure on the base program and then exploits this structure to minimize the amount of work necessary to build the new application.

For a language which doesn't have an incremental compiler, the approach presented here is easier to adopt. Building an incremental compiler is a difficult task although there are attempts to automate it [Hed92]. Adopting this approach requires only that one be able to impose structural constraints on the generated code, and that one use some ingenious code-generation techniques.

A not-so-obvious benefit of the approach presented here is that, in order to generate the extension, and incorporate it into the base program, there is really no need for the source-code of the base program [Mal93a, Mal93c]. This implies that an extension-ready base program can be tailored by its end-user using a tool such as the ApplBuilder, without any need for its source-code. Hence, even end-users can use the ApplBuilder to prototype their applications, or to make changes to prototypes delivered to them.

There is potential for the presented approach to be applicable in other situations. A possible use situation could be an application that generates code from a specification. One could imagine generating an extension-ready base program given a specification. Then every edit operation on the specification could be handled by generating extensions of patterns in the base program. In order to be able to handle all edits of the specification in this way it would be necessary to impose some code-generation constraints on the base program.

The presented approach has been tested by executing the code for the examples presented here. The ApplBuilder hasn't yet been adapted to use this approach, the only reason for this being the lack of resources. All the technology underlying this approach is available to us. It is anticipated this work will be undertaken in the near future. Once that is done, it should be possible to compare the time, between edit and test phases, taken by this approach versus the time taken by the traditional approach.

## 5.5 Related Work

[Mal93a] shows how Beta programs can be made dynamically extensible. [Mal94] presents an approach for the construction of extensible systems in object-oriented languages. These two papers establish the foundations of the approach presented here. The report [Mal93c] describes the implementation of an interpreter that can be used to make this approach work.

[AF89] also show how Beta programs can be made dynamically extensible, but using dynamic linking instead of interpretation. They illustrate the construction of a callable loader; this can be invoked by a Beta program to load and link compiled Beta patterns

into the executing application. This could form the basis of the implementation of the extension server.

There are other direct-manipulation based code generators. Prototyper[6] generates code in a high-level language like C or Pascal. The tool allows, to a limited extent, the linking of user-interface components to each other. This allows one to prototype the behavior of the interface. However, in order to associate any non-trivial behavior with the interface, the generated code must be compiled and linked with the rest of the application. This process is non-incremental.

Prograph[7] is a full-fledged visual programming environment which also offers the possibility of graphical editing of user-interface components. Prograph supports programming in its own special object-oriented language; methods are described as data-flow graphs. The language is dynamic: it is possible to stop execution at a breakpoint, edit methods, and continue execution with the edited methods. As a result, code generated by the user-interface generator can instantaneously be made part of the remainder of the application. This can even happen during the execution of the application.

The Interface Builder[8] generates code in Objective-C. It does not support the incremental generation of new applications when user-interface components have been edited.

In general user-interface generators for dynamic languages such as Smalltalk, CLOS, Self, will not benefit from the approach presented here. This is because it is easy to make changes to a compiled, or even executing, program. Generators based on static languages such as C++, Objective-C, Eiffel, will benefit from this approach.

## Acknowledgements

This work has been generously supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. Thanks to Kaj Grønbæk and Randy Trigg for motivating this problem, and to Kaj for his detailed comments on early drafts of this paper. Thanks also to Ole Lehrmann Madsen for his comments. Thanks to Henry Michael Lassen for his assistance with setting up the examples on the macintosh.

[6] Trademark of SmethersBarnes.

[7] Trademark of Gunakara Sun Systems Limited.

[8] Trademark of NeXT computer.

# Chapter 6

# Building Tailorable Hypermedia Systems: the embedded-interpreter approach

**Author:** Kaj Grønbæk & Jawahar Malhotra

**Date:** January 1993

**Publication Information:** To be submitted as a conference paper.

**Abstract.**

This paper discusses an approach for developing highly tailorable hypermedia systems in an object-oriented development environment. The approach is based on the use of: 1) a hypermedia-development framework with generic classes and objects (DeVise hypermedia-development framework), and 2) an embeddable interpreter for the framework-development language. The approach is based on instantiating the framework into a specific Hypermedia system with the interpreter embedded within the system and uses some reflective techniques. This specific Hypermedia system has a number of "open points" which can be filled via the interpreter at run-time. These "open points" and the interpreter provide sufficient support to allow extensions to the underlying framework at run-time. It is in this way that the specific system becomes tailorable. The approach also supports compile-time tailoring of the hypermedia system. Among the types of tailoring supported are: 1) extension of the hypermedia system with new media-types, 2) alternating editors for supported media-types, and 3) removing a supported media-type from the system. The paper describes the framework and illustrates how the interpreter can be embedded within a Hypermedia system. It describes the steps involved in tailoring

a specific hypermedia system with a new drawing media-type. Drawings contain objects with various graphical shapes; these objects can be anchored as endpoints for links. Since the hypermedia-development framework uses a persistent object-store to store hypermedia documents, issues in handling persistent interpreted objects are discussed; a solution for handling such objects is presented. The paper also discusses how the approach applies to other types of systems, and compares the embedded-interpreter approach with other environments that support tailoring.

## 6.1 Introduction

Recently, terms like 'adaptability', 'customizability', 'extensibility', and 'tailorability' have been used to describe systems that offer their users some potential to change the system. The use of 'tailorability' in this paper covers all of these terms. The term 'tailorability' was coined by Trigg et al. [TMH87] as a by-product of research in the area of hypermedia. They call a system tailorable if it allows users to change the system, e.g. by building accelerators, specializing its behavior, or by adding functionality. The examples given in their work illustrate changes to the behavior of the NoteCards system by using its Application Programmer's Interface (API), and the addition of new media-types by means of the NoteCards card-type mechanism. In this sense, tailoring implies adding and modifying a delivered system at the source-code level; this was possible because NoteCards was built within the residential Interlisp environment [MT81] where any function in the system is accessible, and modifiable, at any time.

In contrast, another extensible hypermedia system Intermedia [Mey86, GS86] was built as a hypermedia-development framework specialized from the general MacApp Application Framework [Sch86]. The Intermedia framework provides support for users (hypermedia designers) to construct their own variants of Intermedia by adding new media-types as specializations of classes from the framework. MacApp was, at that time, written in Object Pascal[1], and such an extension required access to the Intermedia source code, and a compiler, in order to perform the necessary recompilation.

Both NoteCards and Intermedia support tailoring of hypermedia systems, but they both assume that the tailor has the same access to the entire development environment as the original developer had. This assumption is both undesirable and unrealistic in most use settings: the full development environment is incomprehensible, delivering source code raises legal as well as maintenance problems, a full development-environment license is expensive, etc. Thus we face some serious problems: How do we support source-code-level

[1]Now a C++ version is also available.

tailoring of systems without delivering entire development environments? How can such tailorability be provided for compiled-language environments aimed at programming-in-the-large?

This paper discusses an approach to overcome these problems and provide source-code level tailorability, at selected "open points" [Nør92], in systems built with the Mjølner Beta environment. This approach allows users to extend such tailorable hypermedia systems in a fashion similar to that of NoteCards and Intermedia, with the power to do similar things, but without the need for users to have access to the full development environment. Although this paper elaborates the approach in the hypermedia domain, the authors believe that the approach can be applied to other application domains as well.

The DeVise Hypermedia (DHM) system and development framework [GHMS94, GT94, Grø93] formed the basis of these tailorability experiments. The DHM development framework consists of a set of generic classes providing an object-oriented implementation of the Dexter model concepts [HS94]. The Dexter model is a general model that describes the data and functional model of hypermedia systems. The generic classes in the DHM framework provide a conceptual schema for the object structures to be stored persistently, as well as classes to handle the runtime behavior of the link-following and browsing operations. The DHM framework is built using the Mjølner Beta environment which is based on the strongly-typed, block-structured, object-oriented programming language Beta [MMPN93].

The original DHM framework supports development and tailoring of hypermedia systems by writing specializations of the generic classes; this development and tailoring requires access to the Beta compiler and the source code of the system being tailored. Hence the original DHM framework leaves one in a situation similar to NoteCards and Intermedia — there is a need for the development environment and source code of the original system. The situation is rectified by instantiating the DHM framework into a specific system which includes the Beta-interpreter library [Mal93a]. In this system, it is still possible to specialize the generic classes; this is handled by the embedded interpreter. There is no longer any need for the development environment or the complete source code of the system; only the interfaces of the classes being specialized during the tailoring process are needed. As a result, tailorability is transferred from the domain of the framework users (i.e. the developers) into the domain of the system users (most likely super-users with programming knowledge).

The paper explores a scenario in which a user of the DHM system would like to include his/her drawings, made with an independent drawing editor, into his/her hypermedia documents with support for links to the individual elements of a drawing. In other words, the user would, for example, like to be able to link the name of a room in the textual

description of a tour of a house to the corresponding graphical object in a plan of the house. The entire process for accomplishing this is illustrated — this includes the code that the user has to write as well as the user's interaction with the hypermedia system to install this new code.

The paper also resolves the apparent conflict between persistence and extensibility; it shows that an object, which is an instance of an interpreted extension class, can safely be saved into a persistent store or an object-oriented database (OODB) [ABH+92]. It presents a scheme whereby a system which accesses such an "extended object" from the store gets extended automatically, if necessary, with the corresponding extension class.

This work also make concrete the results presented in related papers [Mal93a, Mal94] by implementing a large tailorable system according to the principles in those papers. The tailorable DHM system, built as part of this work, uses the techniques described in both of those papers.

Although the development of the framework and the tailorable system have been done in Beta, every attempt has been made to present the results in a language-independent manner. Hence, where possible, class diagrams are used instead of Beta code. The little Beta code that appears should be self-explanatory if it isn't already annotated with a footnote.

The paper begins by illustrating a tailoring scenario (Section 6.2) in which the user's interaction with the hypermedia system is illustrated. The architecture of the hypermedia-development framework is shown next (Section 6.3). This establishes the background necessary for illustrating how this framework can be instantiated into a tailorable hypermedia system (Section 6.3.4). The code that the user needs to write in order to add the drawing media-type is illustrated in Section 6.4. The scheme for making extended objects persistent is illustrated in Section 6.5. Section 6.6 explores whether an extension made for one version of a system can still work with a newer version of the system. Section 6.7 concludes the paper.

## 6.2 A hypermedia tailoring scenario

This section gives an intuitive feel for the types of extensions supported by the tailorable DHM system. The tailorable DHM system is extensible in that it allows for the introduction of *arbitrary* new media-types. As an example, it is shown how the supported media-types can be extended *dynamically* to allow a new media-type comprising of drawings containing objects with various geometric shapes. The new drawing media-type allows the creation of links to/from the individual objects in a drawing.

A hypermedia system is usually built to work with certain types of media; it may, for example, support text, still pictures, audio, and video. This means that a hypermedia built using the system will have components each of whose contents can be either text, still pictures, audio, or video. For each supported component-type, the hypermedia system usually has built-in support for handling (creating, displaying, storing) the component, as well as for linking to the component.

The DHM system is one such system. Figure 6.1 shows the original non-tailorable[2] DHM system with a hypermedia comprising of two components: a text component and a file component. A text component can contain arbitrary text; links can originate/terminate

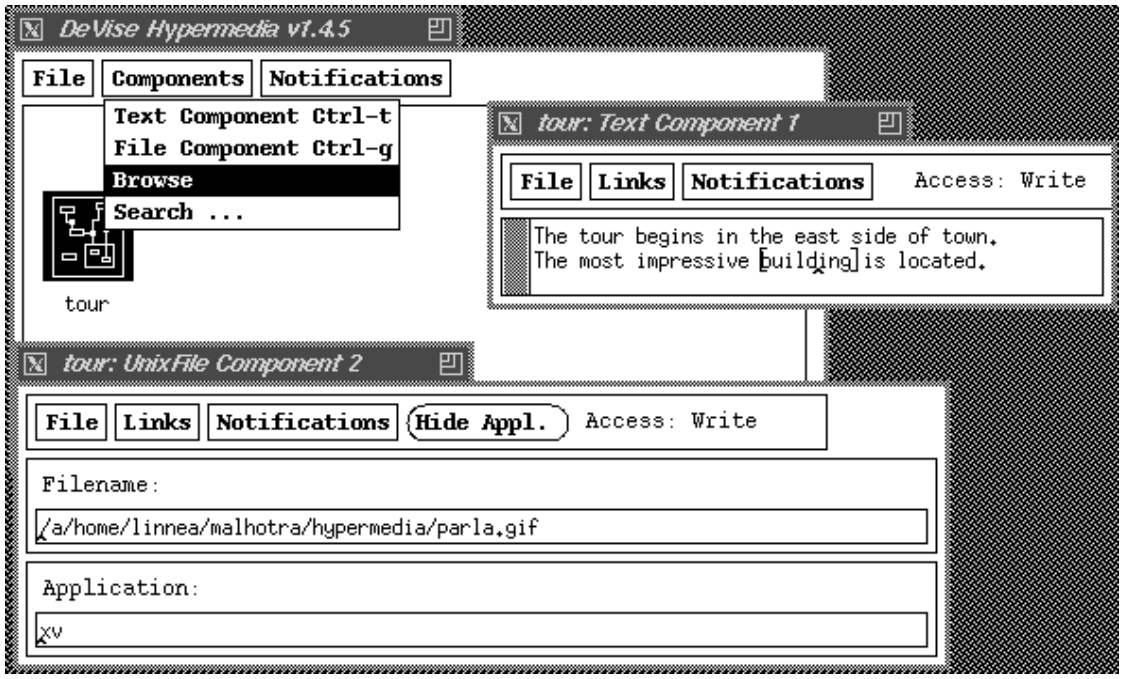


Figure 6.1: The original DHM system with a text component and a file component

within the text component, i.e. substrings can serve as link end-points. A file component contains the names of a data-file and an application-file; activation of the file component (e.g. following a link to it) results in an invocation of the application-file, as a separate process, on the data-file. Links can originate/terminate at the file component, *not* within it. The figure shows the main window entitled Devise Hypermedia v1.4.5. Within this, is shown a hypermedia document called `tour`. The word `building` in the text component is a link end-point. This link terminates at the file component; following it will result in the activation of the file component which will result in the picture-viewing application `xv` being started as a separate process on the file `parla.gif`. A picture will appear on the screen.

[2]non-tailorable means *dynamically* non-tailorable

The file component is both general and limited. Its generality stems from the fact that instances of it can represent arbitrary media-types; the media-type depends on the application-file: if the application supports video, then the component is a video component. This generality comes at the expense of being forced to link to the component *as a whole*; it is not, for example, possible to specify a link to a specific segment of the video sequence.

Thus, the DHM system has, built into it, complete support for the text media-type and limited support, via the file component, for a variety of arbitrary media-types. Arbitrary new media-types cannot be completely supported without modifying the original system and rebuilding it. Hence, if you are a user of the hypermedia system and you have a new media-type you would like to link-in with your hypermedia documents, you will have to get the development version of the hypermedia system, change it, and rebuild it — a task which proves quite expensive and difficult for most users.

In order to support the introduction of arbitrary new media-types, a special tailorable[3] version of the DHM system has been built. This tailorable version has, embedded within it, a Beta interpreter capable of interpreting code for new media-types within the context of the executing hypermedia system. In this tailorable version, new media-types can be defined without any need for modifying or rebuilding the hypermedia system. Given an ordinary editor for drawings, one can introduce a drawing media-type into the hypermedia system by simply writing some code and using the Extend command of the hypermedia system.

The following section illustrates, intuitively, such a tailoring scenario. It illustrates the user's interaction with the hypermedia system during the tailoring process.

### 6.2.1 The drawing media-type extension

It is easy to imagine hypermedia documents with drawings as part of them. It would be nice, for example, to make a plan of a house, and link each room in the plan to a picture of it (as in Figure 6.2). Each room could also be linked to a textual description of it and vice versa.

Assume one has a drawing editor one uses to make drawings; assume also that this editor is available, either as source-code, or as an object-code library. Illustrated here is the procedure by which the tailorable DHM system is extended with this drawing editor, thus allowing one to create hypermedia documents containing drawings, with links pointing to elements within them.

[3] *dynamically* tailorable

Intuitively, assuming that the user has already written some additional code that 'wraps' the drawing editor, this would work as follows:

1. The hypermedia system has an Extend menu-item in its Components menu. The user invokes this item.

2. The hypermedia system prompts the user for the name of a file containing the code to support the new media-type; in this case the user specifies the code for the drawing media-type i.e. the drawing editor and the code that wraps it.

3. The hypermedia system loads the code and adds a new menu-item, called Draw, to the Components menu.

4. The user is free to create instances of the drawing component-class.

A drawing component is illustrated in Figure 6.2: the snapshot shows the Components menu after the drawing component has been added to the system. A drawing component can have an arbitrary number of graphical objects, each of which has some geometric shape. Each of these graphical objects may be an end-point of a link. The figure shows the scenario described earlier: a plan[4] of a house is shown. There are links from the text to the individual rooms in the plan. For example, following a link from the word `kitchen` results in the appropriate room in the plan being highlighted. It is also possible to follow the link backward from the room in the plan to the corresponding word in the text component. In short, the drawing media-type is *completely integrated* with the hypermedia system.

A hypermedia document containing drawing components can also be saved in, and reloaded from, the object-oriented database, just as a normal hypermedia document without any extended drawing components is. Opening a hypermedia document with the extended drawing components in a hypermedia system without the extension results in the automatic incorporation of the extension; i.e. the above user-initiated extension process shown earlier takes place automatically and transparently (see Section 6.5).

As a result of this process, the hypermedia system has been extended with a new media-type, a media-type for which there was no direct built-in support; all of the support was dynamically loaded. The extended hypermedia system will provide complete support for the new media-type with links to and from the internals of the media-type. Installation of the extension requires no change to the hypermedia system; in fact, the extension happens dynamically.

[4] albeit crude!

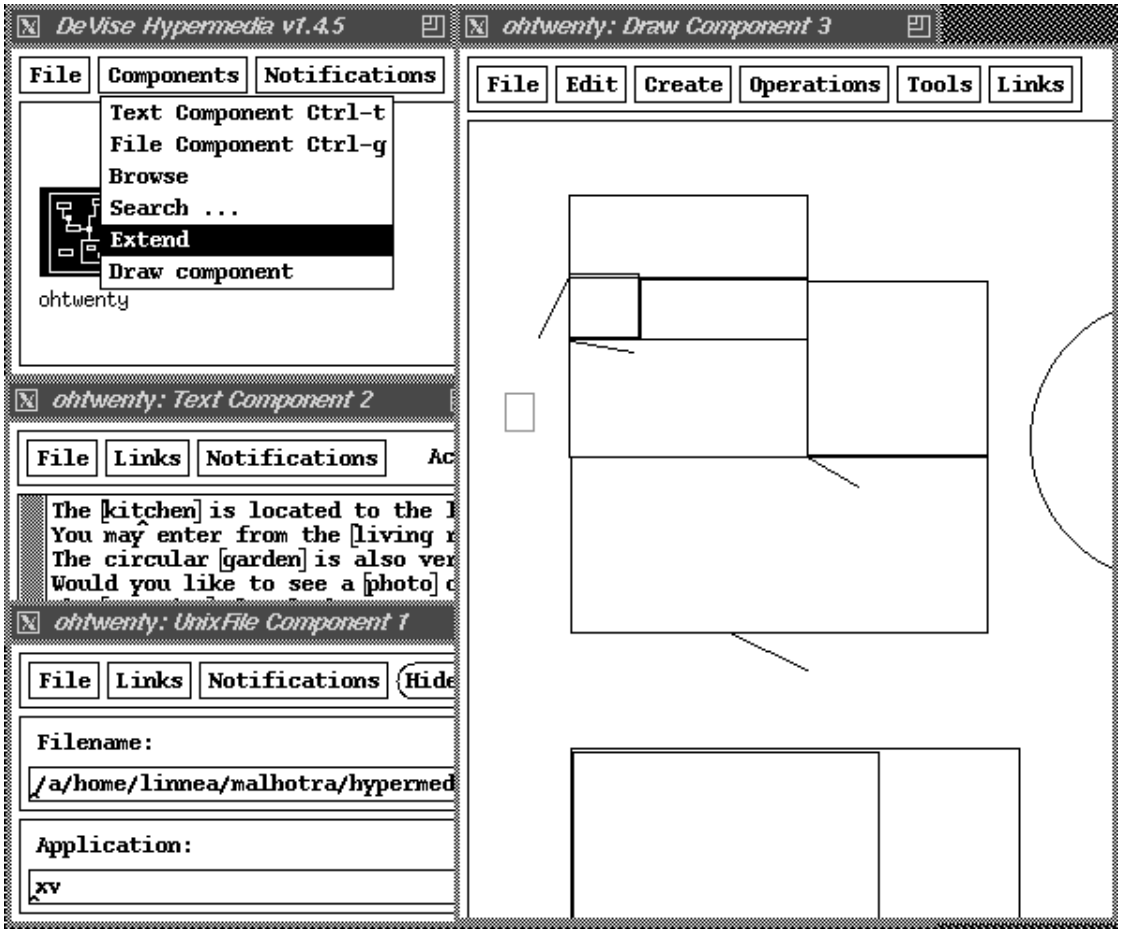


Figure 6.2: The DHM system integrated with a drawing editor

In this manner, the DHM system can be tailored to support arbitrary media types. In order to introduce a new media-type, the user has to:

1. write the code to support the media-type, and
2. perform the above extension-process which essentially loads the support-code into the hypermedia system.

The following sections examine the details of the implementation that contribute to making such an extensibility mechanism work.

## 6.3 DeVise Hypermedia Tailoring architecture

This section presents a brief overview of the DHM tailoring architecture.

### 6.3.1 Framework architecture

The DHM system is built from a Dexter-based hypermedia development framework [GHMS94, GT94]. The architecture of DHM is depicted in Figure 6.3. The architecture is divided into four high-level layers: Storage Layer, Runtime Layer, Presentation Layer, and Application Layer.

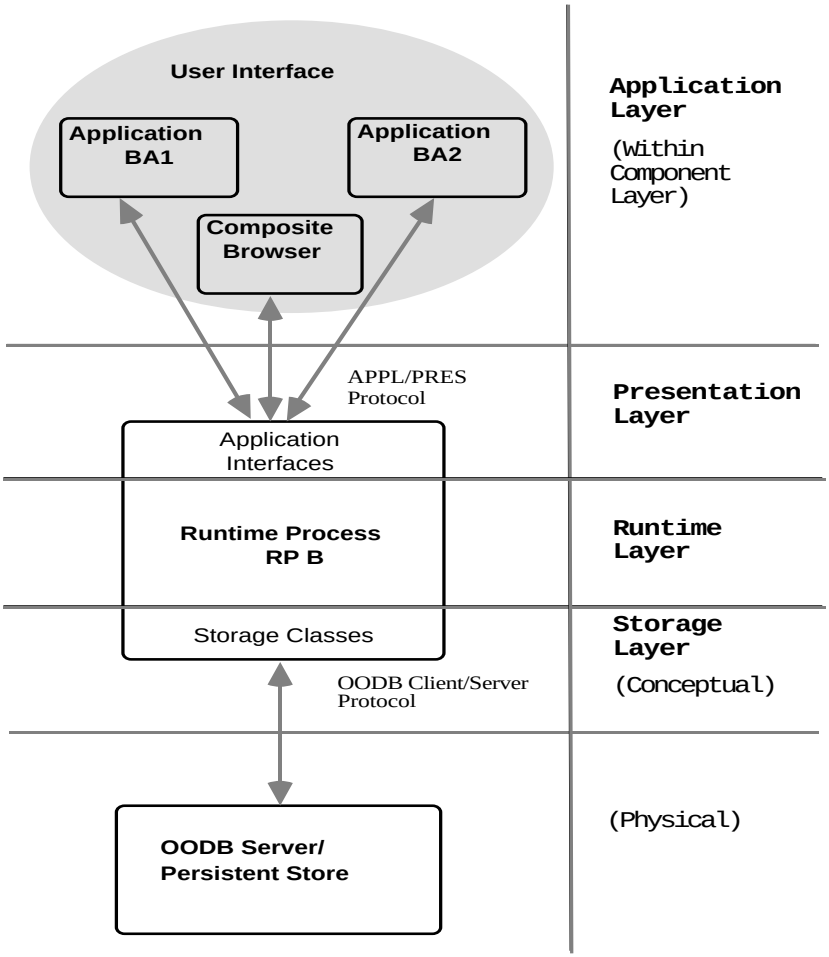


Figure 6.3: The architecture of DHM systems

The Storage layer corresponds to the Storage layer of the Dexter Hypertext Reference Model [HS94] and handles "permanent" data, that is, structural and content information saved in the shared database. The Runtime layer also corresponds to the Runtime layer of the Dexter model and handles information of a transient nature (used only at runtime) as well as supporting hypermedia procedures involving component-creation, link-following, etc. The Presentation layer is a layer introduced to provide an encapsulation of platform-specific code. The Application layer covers the "Within-component" layer of Dexter. The Presentation layer supports access/communication to and from (external) applications

and editors belonging to the Application layer.

The Storage and Runtime layers are written completely in Beta, but are otherwise fully platform independent. The Presentation layer is also written in Beta; its interface is platform independent, while its implementation is platform and application dependent. The Storage layer may be separated into a client/server configuration (using the OODB; see Figure 6.3) or built into the runtime process (using the persistent-store library).

The Presentation layer defines a protocol for communication with the Application-layer process. The protocol is implemented via direct invocation for applications built into the runtime process; for external applications written in Beta, it is implemented with the BETA distribution facilities; it may use sockets on Unix, AppleEvents on the Macintosh, and DDE on MS-Windows to support communication with non-Beta external applications. Thus the Application layer may be written in any language that can support the implementation of a communication protocol with the Beta code in the Presentation layer.

So, one possible configuration for the system as a whole is with the Application-layer applications built into the runtime process and the Storage layer separated into a client/server configuration (OODB). In this case the APPL/PRES protocol boils down to direct method-invocation. This is the configuration used for this work.

### 6.3.2 Framework Classes

The underlying development framework consists of sets of classes to implement the three within-hypermedia layers: Storage, Runtime, and Presentation (see Table 6.1).[5]

It is beyond the scope of this paper to explain all the details of the framework as outlined in Table 6.1; however, in order to give an idea of how one can develop specific systems, or extend existing systems, a brief description of the overall framework is needed.

#### Storage layer classes

The Storage-layer generic classes, plus specialized classes, constitute the conceptual schema for the objects that are stored in the Persistent Store/OODB. In the Storage layer, the StorageMgr class manages the storage of Hypertext objects. A Hypertext object holds a set of components. A Component is an abstraction that implements both "nodes"

[5]Notation: Indentation indicates block-structural nesting of classes. Class names in plain text describe generic classes, whereas names in italics describe examples of specialized or application-specific code that can be added to develop a specific hypermedia system. Names in angular brackets (e.g. `<list-of-Component>`) denote the attributes of the classes they appear within.

| Storage Classes | Runtime Classes | Presentation Classes | *Application Layer* |
|---|---|---|---|
| | | Presentation<br><Appl-ref><br><Protocol> | |
| StorageMgr | SessionMgr<br><StorageMgr-ref><br><sessionMgrPres-ref><br><list-of-Session> | SessionMgrPres<br><SessionMgr-ref> | *(DHM mainwindow)* |
| Hypertext<br><list-of-Component> | Session<br><Hypertext-ref><br><sessionPres-ref><br><list-of-Instantiation> | SessionPres<br><Session-ref> | *(Hypertext icon in DHM mainwindow)* |
| Component<br><BaseComponent-ref><br><list-of-Anchor><br><PresSpec-ref> | Instantiation<br><Component-ref><br><instPres-ref><br><list-of-LinkMarkers> | InstPres<br><Instantiation-ref> | *(Editors for component contents)* |
| Anchor | LinkMarker<br><Anchor-ref><br><LinkMarkerPres-ref> | LinkMarkerPres<br><LinkMarker-ref> | *(e.g. brakcets around text in a TextEditor)* |
| BaseComponent | | | |
| LinkComponent<br><list-of-specifier> | LinkInstantiation<br><linkComponent-ref> | | |
| Specifier<br><Component-ref><br><Anchor-ref><br><PresSpec-ref> | | | |
| CompositeComponent<br><list-of-Component> | CompositeInstantiation<br><compositeComp-ref> | | *(XIconBrowser)* |
| | SubInst<br><Component-ref><br><SubInstPres-ref> | SubInstPres | *(Icon in XIconBrowser* |
| AtomComponent<br>*(TextComponent FileComponent etc.)* | *(TextInst FileInst etc.)* | *(TextPres FilePres etc.)* | *(TextEditor FileWindow etc.)* |

Table 6.1: An outline of the relationship between framework classes

and links of traditional hypertext systems. A component holds some contents and one or more anchors, each of which identify a location within the component's contents (e.g. a sub-string in a text component). In a component object, the contents are represented by a pointer to a BaseComponent object, and the anchors by a list of Anchor objects.

The LinkComponent sub-class implements links, the AtomComponent sub-class implements "nodes" with simple data contents like text, the CompositeComponent sub-class implements "nodes" comprising sets of other components, or having some internal (hierarchical) structure. TextComponent and FileComponent are examples of specific AtomComponents added to a specific hypermedia system. CompositeComponents are, for example, used to contain sets of AtomComponents resulting from a query search.

### Runtime layer classes

In the Runtime layer the SessionMgr class manages sessions with multiple hypertexts. The Session class possesses the runtime behavior for a hypertext; hence it has pointers `<hypertext-ref>` to a hypertext object, and `<sessionPres-ref>` to a session-presentation object. See the following section for an explanation of presentation objects.

The Instantiation class which is nested within the Session class implements the runtime behavior of components. It has a pointer to a component object from the Storage layer, and to an instPres object from the Presentation layer. The instPres object is a wrapper that provides a uniform interface to editors for various component contents (media-types). The linkMarker class implements the runtime behavior of anchors; a linkMarker object has a pointer to an anchor object and to a linkMarkerPres object. Presentation objects are explained in the following section.

### Presentation layer classes

In the Presentation layer there is a generic Presentation class which acts as a superclass for all presentations; it possesses a reference to an editor or some other user-interface object of the Application layer. The Presentation-class attributes represent the general protocol for communication between the user-interface and the within-hypermedia objects. The sessionMgrPres class wraps the user-interface (i.e. the main DHM window) used to access the OODB and to start and close sessions on specific hypertexts.

Similarly, the sessionPres class wraps the user-interface to a specific hypertext (e.g. the icon representing the hypertext which appears in the DHM main window, and a collection of menu-items to operate on it). The instPres class wraps an editor for a component's contents. Finally, the LinkMarkerPres class which is nested in InstPres wraps the user-interface code responsible for presenting an anchor in the component's contents, e.g. brackets in the text belonging to a text component.

### Class organization

The classes of the different layers are organized in inheritance hierarchies; a specific hypermedia system has its own extended specialization hierarchies mirroring the media-types and composite-types that are provided in that particular system. Figure 6.4 shows an example of the Component class-hierarchy for a given system supporting only text and external files as atomic media-types, along with a rich variety of composites to handle browsers, and queries [GT94]. Similar specialization hierarchies exist for the Runtime and Presentation layers.

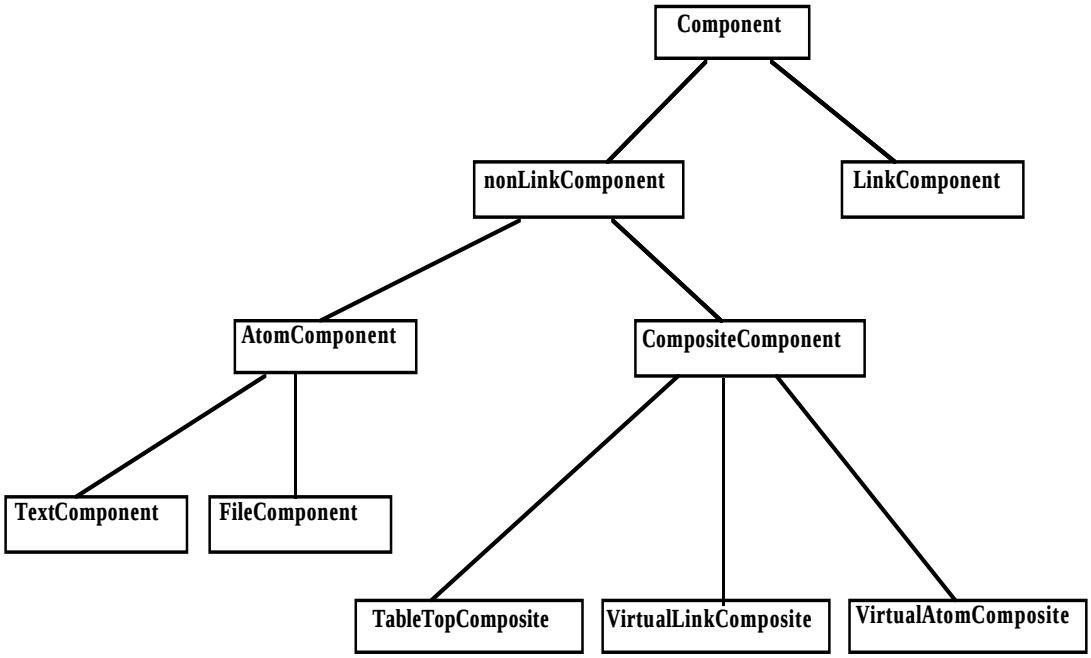


Figure 6.4: Example of original DHM Component inheritance-hierarchy

### 6.3.3 Framework support for a new media-type

The purpose of the hypermedia development framework is to support the rapid construction of hypermedia systems aimed at different application domains. The generic framework-classes provide extensive, and general, hypermedia functionality which can be reused in many different domains by developing specialized classes suiting the needs of the domain.

The framework has been organized in such a way that a hypermedia designer can bring in a new media-type by providing an editor for the new media-type and by building his/her own specialized classes for each of the Storage, Runtime and Presentation layers. Having built his/her classes he needs to register them in the sessionMgr's type-table. This registration is done by means of so-called typeInfo objects; a typeInfo object is a kind of meta-object containing information about a particular class. Each of the specialized classes is encapsulated within a corresponding typeInfo object along with additional information about the class. This typeInfo object is then registered in the sessionMgr's type-table.

The typeInfo objects exist in parallel inheritance hierarchies, such that e.g. a textComponent's typeInfo is a specialization of an AtomComponent's typeInfo. The generic typeInfo class is outlined in Table 6.2.[6]

[6]The code is presented in pseudo-Beta syntax. Patterns are a general abstraction mechanism in Beta:

```
typeInfo: Class
(#  attributes: attributeValueTable
    hasAttr: (# attr: Text; found: bool; enter attr do ... exit found #);
    getAttr: (# attr: Text; val: Object; enter attr do ... exit val #);
    setAttr: (# attr: Text; val: Object; enter (attr,val) do ... #);
    removeAttr: (# attr: <text-ref>; enter attr do ... #);
    init: virtual (# do ...#);
#)
```

Table 6.2: Pseudo-beta description of the typeInfo class

The typeInfo class contains an attribute-table consisting of (name,value) pairs; there is a general procedural interface for manipulating this table. This makes all typeInfo objects look similar to the sessionMgr; only the actual registration of attributes varies throughout the inheritance hierarchy. The attributes of a typeInfo object are typically set during initialization, but they may also be set dynamically at any time. An example of initializing the typeInfo object for the DrawComponent extension discussed in this paper is shown in Table 6.3.[7] In the DrawCompTypeInfo object, a name and a reference to the class DrawComponent##, among other things, are registered.

The Mjølner BETA environment supports classes as first-class objects in a program; hence the typeInfo objects can point to the classes they describe. This makes it possible to maintain a database of registered types, and query it for different kinds of information related to the class. This table is used, for example, to compute the contents of the Components menu (see Figure 6.1): the typeInfo table is queried to return names and pointers to classes for all the components that are setup to be visible via the Components menu; for each such component, a menu-item is created. The menu-item is setup to use the corresponding class-pointer to create an instance of the component.

Another example is when a given component has to be presented, e.g. as the result of a followLink operation; the typeInfo object of the component holds information about the preferred instantiation and presentation types and this information is used to create Instantiation and Presentation objects of the right types.

---

they unify classes, types, procedures, and functions. `typeInfo` is a class pattern, `attributes` is a static attribute of the class, `hasAttr`, `getAttr`, `setAttr`, `removeAttr` are procedure patterns (methods) of `typeInfo`, `init` is a virtual procedure-pattern (virtual method) and hence may be further bound in specializations of `typeInfo`. In procedure patterns, the enter-part declares the formal parameters, the do-part declares the body, while the exit-part declares the return value.

[7] Here `DrawCompTypeInfo` specializes `AtomCompTypeInfo` and further binds its *init* virtual procedure-pattern (virtual method). Assignment is expressed by `->`: the value on the left is assigned to the entity on the right. Patterns with the `##` suffix denote pattern values, i.e. patterns as first-class values.

```
DrawCompTypeInfo: Class AtomCompTypeInfo
(# ...
   init: binding
   (#
     do ('Name','DrawComponent') -> setAttr;
        ('Class',DrawComponent##) -> setAttr;
        ('PrefInstTypeName','DrawInstantiation') -> setAttr;
        ...
   #);
#)
```

Table 6.3: Pseudo-beta description of the DrawCompTypeInfo class

To add the drawing media-type to the hypermedia system with the component hierarchy as depicted in Figure 6.4, a hypermedia designer needs to provide the specialized classes shaded gray in Figure 6.5. Having developed these classes, the corresponding typeInfo objects only need to be registered with the sessionMgr's typeInfoTable. The drawing editor automatically gets included into the hypermedia system, and the drawing component becomes available to the user via the Components menu.

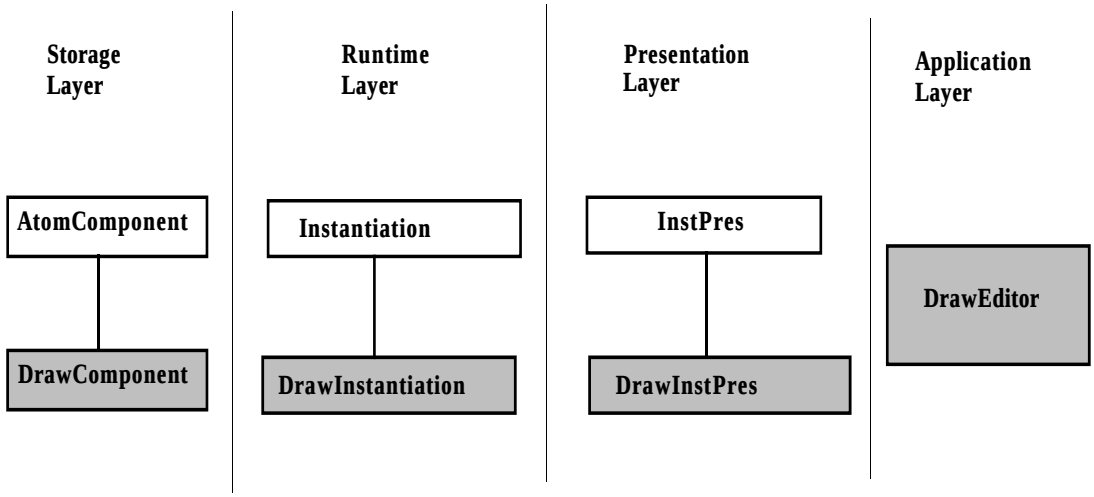


Figure 6.5: Classes involved in adding a Drawing media-type to the hypermedia system

### 6.3.4 Instantiating the framework into a tailorable DHM system

This hypermedia-development framework allows for the addition of new media-types by simply registering the required classes and rebuilding the system. This process requires the addition of code to register the new classes, compilation of this additional code and of the new classes, and a relink of the system.

The tailorable DHM system is built from this framework by including the Beta interpreter [Mal93a]. The embedded interpreter is capable of dynamically incorporating new classes into an executing hypermedia system. The tailorable DeVise hypermedia system uses the interpreter to load the classes for a new media-type and then registers them with the sessionMgr's typeInfoTable. Thus, the extensibility which was originally available at the framework level, is now available within the tailorable DHM system without any need for recompilation and relinking.

The tailorable DHM system contains the code presented in Table 6.4;[8] this code invokes the interpreter and registers the returned classes in the typeInfoTable.

Assuming the user has provided a name for the new component (`compName`) along with the names of the files containing the Beta code for the presentation layer (`presFile`), the instantiation layer (`instFile`), and the storage layer (`compFile`), the above code invokes the interpreter on each of these files. The interpreter returns a class-pointer to the corresponding `typeInfo` class. These are instantiated and the resulting `typeInfo` objects are stored in the `sessionMgr`'s tables. These objects are then initialized, a process that sets up the information in these objects. Finally, the components menu is recomputed.

In the tailorable DHM system, when the user invokes Extend, this code gets executed. Recall the tailoring scenario described in Section 6.2: the above code gets executed for the drawing media-type. The Draw menu-item then appears in the Components menu (Figure 6.2). When this menu-item is selected, the `sessionMgr` looks in its `typeInfo` table, finds the `DrawCompTypeInfo` object and gets all the information, like the pattern value for the `DrawComponent` pattern, necessary to create a drawing component.

Note that it is possible to define variations of this. One could, for example, write a variant which only changes the presentation of a given component. It is also possible to support the removal of media-types: a simple way is to remove the corresponding menu-item from the components menu, but one could also imagine an open point via which the user could

[8] The code is presented as a Beta procedure-pattern. Variables with the `##` suffix are pattern variables and can hold patterns values. The `&` means create a new instance and the `[]` suffix means take a reference to the instance.

```
extend:
(#
 enter (compName, presFile, instFile, compFile)
 do (* interpret the source code; get the typeInfo classes *)
    presFile -> interp -> presTypeInfo##;
    instFile -> interp -> instTypeInfo##;
    compFile -> interp -> compTypeInfo##;

    (* instantiate the typeInfo classes;  *
     * store typeInfo objects into tables *)
    &presTypeInfo[] -> presTI[]
         -> theSessionMgr.sessionMgrTypes.append;
    &instTypeInfo[] -> instTI[]
         -> theSessionMgr.currentSession.sessionTypes.append;
    &compTypeInfo[] -> compTI[]
         -> theSessionMgr.sessionMgrTypes.append;
    (* instantiate the typeInfo objects *)
    presTI.init;
    instTI.init;
    compTI.init;
    (* recompute the components menu from *
     * the sessionMgr's typeInfo tables   *)
    theComponentsMenu.recompute;
#)
```

Table 6.4: Extending the hypermedia system with an interpreted media-type

remove the corresponding `typeInfo` objects from the `sessionMgr`'s tables. See [Mal93b] for details on how extensions can remove functionality.

## 6.4 The code for the drawing media-type

The previous section showed that in order to add a new media-type to the tailorable DHM system, one has to write various classes for the new media-type and call Extend. This section illustrates the code that needs to be written; it uses the drawing media-type as an example. In Figure 6.5 it was shown that in order to add the drawing media-type to the hypermedia system, it is necessary to define the classes DrawComponent, DrawInstantiation, DrawPres, and DrawEditor. This section focuses on the DrawPres and DrawEditor classes

as these comprise the interface between the drawing editor and the hypermedia system, and are sufficient to illustrate the idea.

The construction begins with a drawing editor. For this experiment, an existing drawing editor was used. This editor supports the creation of graphical objects with various shapes. These graphical objects have identities. There is the notion of a current graphical object; the editor can indicate the current object visually by drawing it in a different color from the other objects. The entire drawing editor is written in an object-oriented style in Beta. The relevant details of the code are examined in the following sections.

Figure 6.6 summarizes the construction process. The existing drawing editor (DrawEditor) is specialized into a presentation-compatible drawing editor (DrawEditorForPres). DrawInstPres, the presentation class for the drawing media-type, is defined as a specialization of InstPres.

When a drawing component is created by the user, an instance of the drawing-presentation is created. This, in turn, creates an instance of the presentation-compatible drawing editor. The DHM object invokes operations on the drawing-presentation object and vice versa; the drawing-presentation object invokes operations on the presentation-compatible drawing-editor object and vice versa. For each drawing component in the hypermedia

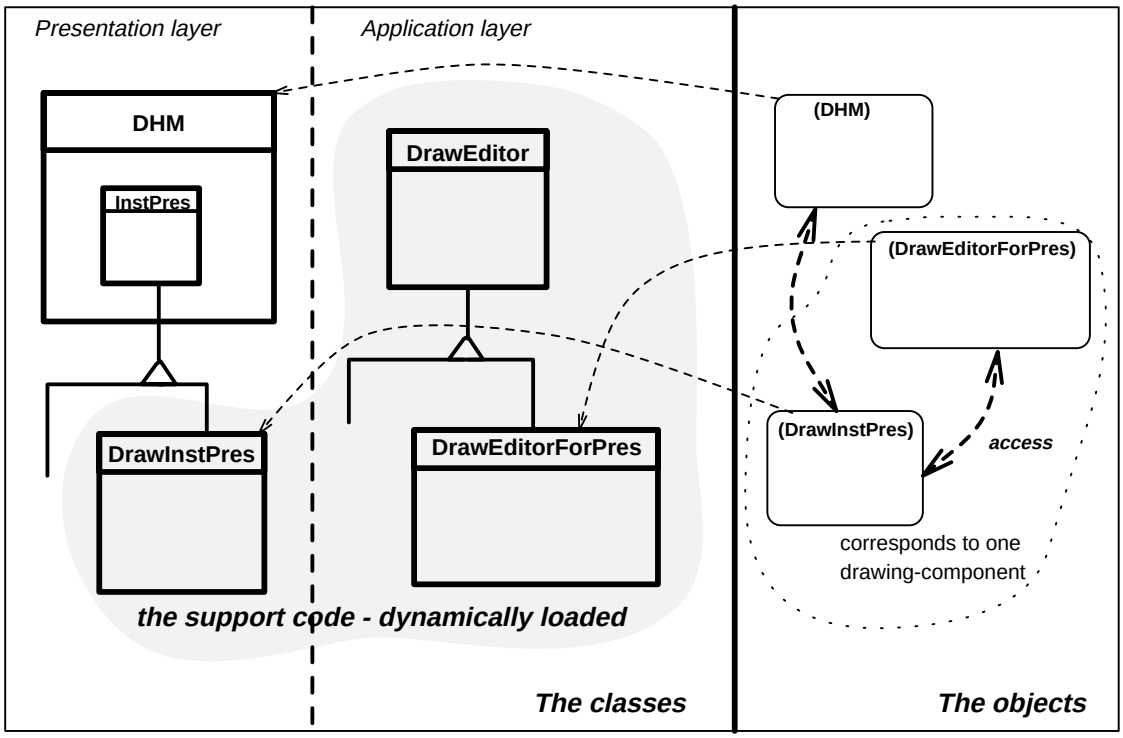


Figure 6.6: The drawing editor and its presentation; see figure 6.7 for class details

system, there will be one DrawPres instance and one DrawEditorForPres instance.[9]

[9]also one instance each of DrawInstantiation and DrawComponent, but these are not discussed here.

### 6.4.1 The details

As shown in Figure 6.6, the construction has two parallel threads: the existing editor is specialized to make it compatible with the hypermedia system and the presentation class in the hypermedia system is specialized to make it compatible with the extended editor.

Figure 6.7 shows the Beta class DrawEditor; this is the original drawing editor. Nested within it are two virtual-classes: drawingSurface and menuBar. The drawingSurface implements the window in which the drawings are made. Nested within it, and not shown in the diagram, are various classes and methods which describe the different graphical shapes and their behaviors. menuBar describes the implementation of the menu bar: all the menus and the items that comprise them are described within menuBar.

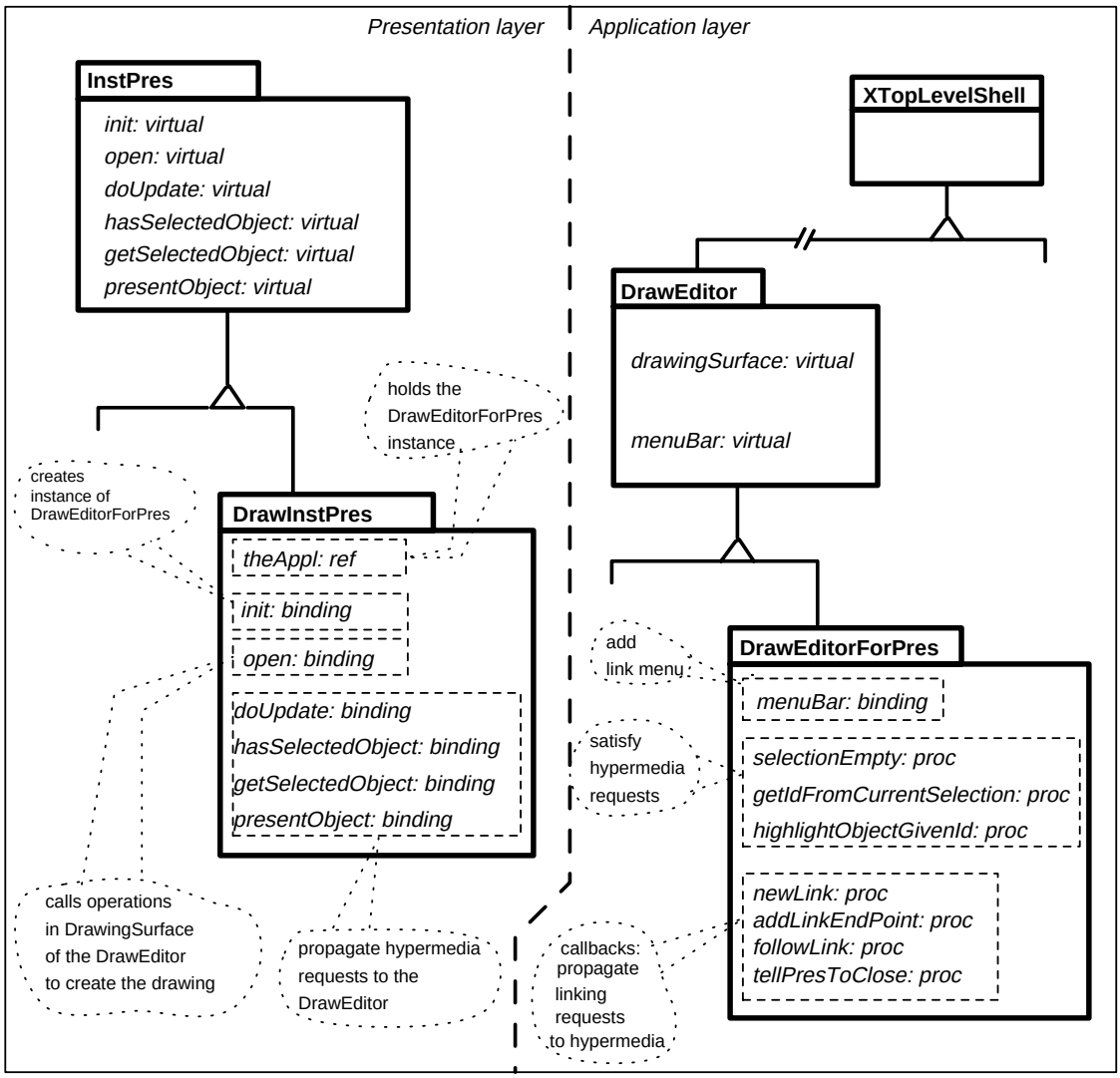


Figure 6.7: The support code for the drawing media-type

DrawEditor is specialized into DrawEditorForPres. In the specialization of the editor, a new

menu called Link is added. This is to enable linking to and from the various graphical objects in the drawing editor (see Figure 6.2). It has items such as New Link Source, Add Link End-point, and Follow Link. The Quit menu-item in the File menu is replaced by a slightly different Quit Window. This is because the editor is no longer a stand-alone application; it cannot just quit as it did before; instead it must inform the hypermedia system that it is about to close and must save its contents within the hypermedia document.

These changes to the menus of the original drawing editor are accomplished by extending the menuBar virtual-class.[10] The new menu commands are implemented in terms of calls to the methods newLink, addLinkEndPoint, and followLink[11] which are defined in DrawEditorForPres.

These methods implement their behaviors by calling operations declared within the hypermedia system. For example, newLink calls thisSession.newLink where thisSession is an attribute declared within the class InstPres (actually a super-class of it). The name thisSession is visible within DrawEditorForPres because DrawEditorForPres is placed lexically within DrawInstPres,[12] thus making it in the lexical scope of declarations made within DrawInstPres and all its super-classes. This is an example of the power of the approach: the extension can reach into the original system and access any state-information and operations visible in its scope.

The Quit Window menu-item of the extended editor uses the tellPresToClose method, declared within the extended editor, which is implemented by a call to myInst.tellPresToClose. Here, myInst is once again a variable declared in a lexically-enclosing block. The tellPresToClose method in myInst calls doUpdate in DrawInstPres. This scans all the graphical objects in the current drawingSurface and writes them into the OODB as the contents of this drawing component.

The methods selectionEmpty, getIdFromCurrentSelection, highlightObjectGivenId are also defined within DrawEditorForPres. These are implemented in terms of simple operations available within the drawingSurface; they get called by the presentation object.

Figure 6.7 also shows the DrawInstPres class as a specialization of InstPres, which is built into the hypermedia system. This is an example of an extension which specializes a built-in class. In fact, in the tailorable DHM system the presentation object is partly compiled and partly interpreted: the behavior inherited from InstPres is compiled while that defined in DrawInstPres is interpreted.

[10] In Beta, virtual sub-classes refine, not replace, their super-classes; they are referred to as further bindings; hence the qualification *binding* in the diagram.

[11] These are Beta patterns used as procedures (methods); hence, the qualification *proc* in the diagram.

[12] the lexical placement is not shown in the diagram.

DrawInstPres doesn't add new operations to the interface of InstPres; it merely defines the implementation of the operations already defined in InstPres. The init method is defined to create an instance of the DrawEditorForPres and to store that instance in the reference-variable theAppl.

The open operation gets invoked by the hypermedia system when an existing drawing component in the hypermedia is to be displayed. Its purpose is to make the editor display the stored drawing. This drawing is stored in the OODB (by doUpdate). The open operation accesses the drawing elements from the OODB and calls operations in the drawingSurface of the current drawing editor to display them.

The doUpdate, hasSelectedObject, getSelectedObject, and presentObject operations form the general protocol for the hypermedia system to get information from the drawing editor.[13] They are implemented in terms of operations selectionEmpty, getIdFromCurrentSelection, and HighlightObjectGivenId defined within the editor.

These definitions are sufficient to interface the editor to the hypermedia system.

### 6.4.2 Typical execution sequences

Say the user creates an object in the drawing editor. The creation events are routed to the drawing editor's event handler via the window-system toolkit. Hence, the drawing editor behaves just as it did outside the hypermedia system.

Now, say the user selects NewLink[14] from the Links menu of the drawing editor. This is handled first by the drawing editor; it calls the newLink method defined within it, which calls the newLink operation in the current-session object in the hypermedia system. This queries the drawing editor, via the corresponding presentation object, for the currently-selected object in the editor. Given this object (or its ID), it creates a link originating within this drawing component at the selected object.

Say the user follows a link to an inactive drawing component (say, from a text component). The link contains a reference to the contents of the drawing component and to a single object within it. The hypermedia system first creates an instance of the drawing presentation (DrawInstPres) and initializes it; this results in a drawing editor. It then accesses the contents of the drawing component from the OODB, calling open on the presentation object. This results in the display of the drawing component contents in the drawing editor. It then asks the presentation object, via the presentObject method, to display the particular object at the link end-point.

[13] and, in general from any editor.

[14] Assume the user is creating the source of a new link.

This section has illustrated the type of coding effort required in order to extend the hypermedia system with a new media-type. Although shown for the drawing media-type, the techniques apply equally well for other media-types. For example, to extend the system with a video media-type, the video editor/viewer would have to be extended to add the link menu, and a VideoInstPres presentation class would have to be defined. VideoInstPres would have the same interface as DrawInstPres; it would just be implemented differently. The presentObject operation, for example, could call an operation in the video editor to show the desired video sub-sequence or frame.

## 6.5 Extensions and Persistence

The DHM system uses the persistent store (or the OODB) to store hypermedia documents. When the system is extended with a new media-type, like the drawing media-type, interpreted drawing components will be part of the saved hypermedia documents. Hence, it must be possible to save these interpreted objects into the store as well as reload them from the store.

When an object is saved into the persistent store (which could be the OODB), a reference to its prototype (behavior) is saved along with its state. A prototype, in Beta terminology, is a runtime descriptor of the class. When the object is loaded into an executing program — which could be another program — this reference is used to locate the prototype of the object in that execution. Normally, a program saving the object has its prototype compiled into it, and a program loading the object is expected to have the prototype compiled into it.

What happens, then, in the case of an object whose prototype is interpreted, or in general, dynamically loaded? When saving the object, the prototype will generally be present in the execution. But, when loading it, its prototype may or may not be available. If the extension has already been installed, then its prototype will be there, else it won't. So, the question is: what happens when an object is loaded and its prototype is not present in the execution?

A satisfactory answer is: load the prototype dynamically, automatically and transparently. Shown here is an approach for making this work. The approach assumes that the extensions are interpreted, but it should work, with slight modifications, even if they are dynamically linked.

In order to explain the special approach, it is necessary to introduce some of the basic functionality of the persistent store. When an object is saved, a reference to its prototype is saved by saving the name of the object file (module) in which it was defined, along

with an index which uniquely identifies the prototype within that file. When the object is accessed, possibly in another program, the name of the object file and the index are used to locate the prototype in the accessing program's current execution's address-space. The name is used to locate the object file in the accessing program, and the index to locate the prototype within that file.

This simple scheme is implemented by ensuring that every program using the store has a table describing all its object files. This table is built when the store is initialized. Intuitively, this table has one entry for each object file in the program it resides within. Each entry describes the corresponding object file; it allows a prototype to be mapped into a unique index and vice versa.

Figure 6.8(A) shows this table, its entries, and the relevant classes; only relevant details are shown. The table is an instance of the class ExecGroupTable (EGT). The entries are instances of CompExecGroupTableElement (CompEGTElement). The table has a method findByName, used to locate an entry by the name of the object file. The first step in saving or accessing an object is to determine which file it belongs to, and then call findByName on the table to access the corresponding entry. Once the entry has been obtained, its methods protoToIndex and indexToProto can be used to get the unique index for the object's prototype (saving), or to get the prototype corresponding to the unique index (restoring).

### 6.5.1 Handling persistence for interpreted classes

A program which wishes to save interpreted objects into the persistent store must use a specialized version of the table. This is illustrated in Figure 6.8(B). The specialized table is an instance of InterpEGT, which is a sub-class of EGT. In this specialized version, the findByName method is extended to handle unsuccessful searches, i.e. searches for files that its super-class cannot handle. Its super-class will be able to handle all searches for files that are compiled, leaving only the interpreted ones to it.

This specialized findByName is designed to invoke the interpreter on the file being searched for. This process creates prototypes, in the current address-space, for all the classes declared in that file. findByName then creates a specialized entry representing that file and returns it. The specialized entry is an instance of InterpEGTElement, which is a sub-class of EGTElement. In order to record the fact that this file has been interpreted, and to be able to service future requests for the same file, findByName also saves this entry in the table, separated from the compiled entries. This is illustrated in Figure 6.8(B), where the table has two sets of entries.

The protoToIndex method of this specialized entry behaves just as it does in normal entries

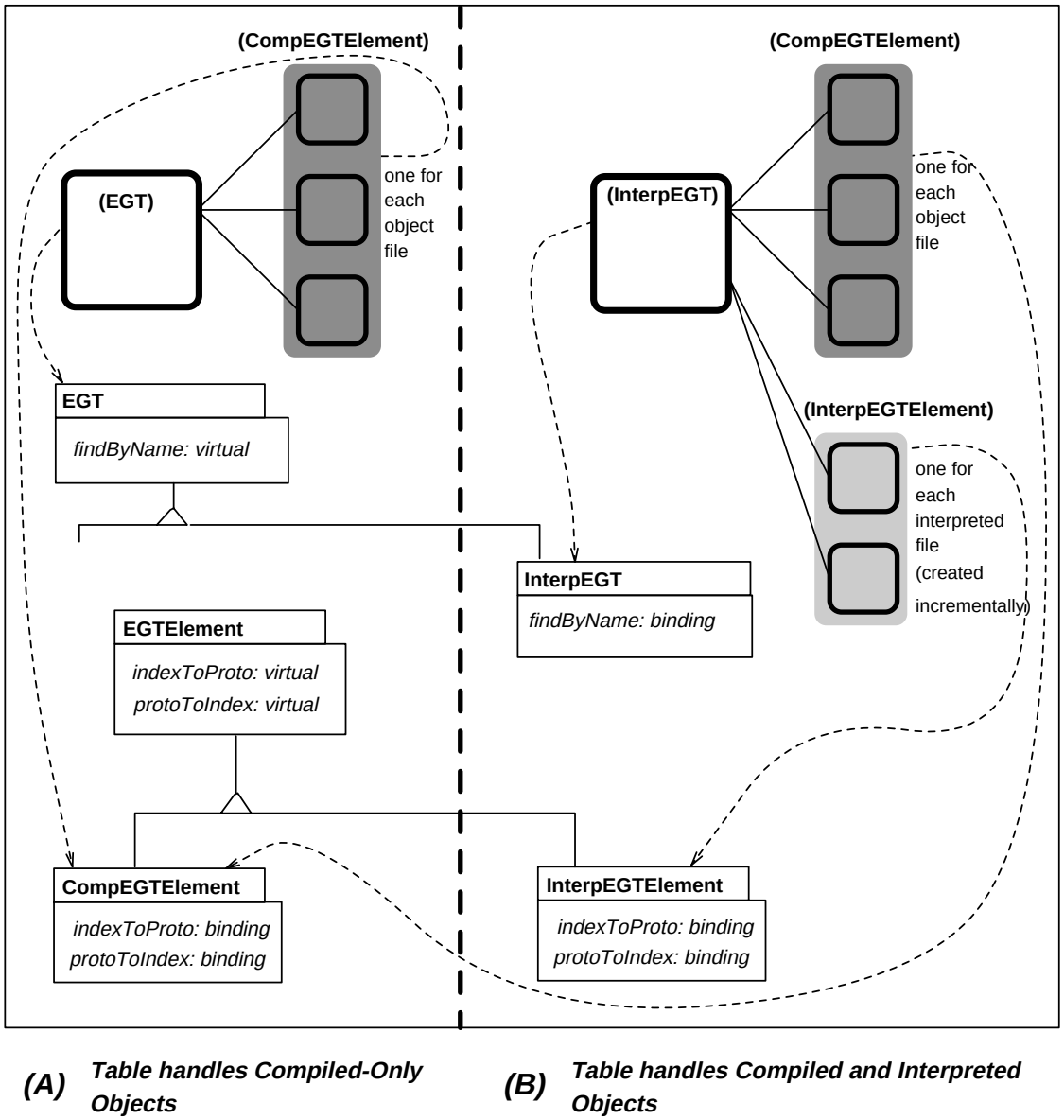


Figure 6.8: The tables for the persistent store

(i.e. instances of CompEGTElement). It maps the given prototype into an index unique within that file. This is done for interpreted files just as it is done for compiled files. The indexToProto method determines where the prototypes for this interpreted file reside in the current address-space. They are guaranteed to be there as findByName has invoked the interpreter on the file. It then uses the index to get the address of the specified prototype.

When an interpreted object is saved, the table is searched, as normally, for the file-name of the object. This is not found by the original findByName; so its extension is invoked. It creates a special entry for the file, if it doesn't already exist, and returns it. Calling protoToIndex on this entry returns an index as before, and the object is saved with its

file-name and its index.

When this object is accessed from the store, the stored file-name is used to search the table in the current process. Again, the original findByName is unsuccessful; hence its extension is invoked. Assuming this file has not already been interpreted, the extended findByName interprets the file, creates an entry for it, and returns the entry. When indexToProto is called on this element, the address of the indexed interpreted prototype will be returned.

Now, assume that this object is accessed from an executable in which its file has been compiled. In this case, an entry for the file-name will be found by the original findByName, and the object will get loaded as a compiled object. Hence, it is possible to save an interpreted object in the store and load it later as compiled object.

Observe that in order to make this approach work, the only parts of the original persistent store that have been specialized are the EGT and EGTElement.

### 6.5.2 Loading persistent interpreted components in the tailorable DHM system

Returning to the DHM system, when a hypermedia document containing a drawing component is saved, only the *component* object (instance of DrawComponent) is saved in the store. The presentation and instantiation objects are not saved.

When this hypermedia document is later accessed, loading the drawing component object triggers a process by which, as described above, the drawing-component class (DrawComponent) — and its typeInfo class, which is defined along with it — will get interpreted and transparently installed into the system. The instantiation and presentation classes, DrawInstantiation and DrawInstPres, will, however, not get defined via this mechanism. These are necessary in order to handle the drawing-component object at run-time. The loading of the code for these classes should also be transparent to the user of the hypermedia system.

To do this, the runtime layer, i.e. the sessionMgr class, is extended to load these classes dynamically. This is handled by requiring that the user write an extended version of the component's typeInfo. Table 6.5 illustrates such an extended typeInfo for the drawing component. Note that this is an extended version of the DrawCompTypeInfo shown in Table 6.3. Here, the user has also specified 'InstPath' and 'PresPath' attributes; these specify the locations of the files containing the source-code for the instantiation and presentation classes respectively.

When the sessionMgr loads a drawing component from the store, the automatic mechanism described above loads the source-code for the drawing component and defines the

```
DrawCompTypeInfo: Class AtomCompTypeInfo
(# ...
   init: binding
   (# do ('Name','DrawComponent') -> setAttr;
         ('Class',DrawComponent##) -> setAttr;
         ('PrefInstTypeName','DrawInstantiation') -> setAttr;
         ('InstPath', '/users/kgronbak/DHM/v1.4.5/UserRUN/DrawInst.bet')
             -> setAttr;
         ('PresPath','/users/kgronbak/DHM/v1.4.5/UserPRES/DrawPres.bet')
             -> setAttr;
        ...
   #);
#)
```

Table 6.5: The DrawCompTypeInfo class with information about the source-code location of the presentation and instantiation classes

classes DrawComponent and DrawCompTypeInfo. The sessionMgr then creates an instance of DrawCompTypeInfo and initializes it. It can now access the 'InstPath' and 'PresPath' attributes of this object and interpret the respective files. This process will define the necessary presentation and instantiation classes. The loaded drawing component object is now ready to be used.

With this scheme in place, the `extend` pattern described in Table 6.4 can be modified in order to obtain `presFile` and instFile from the compTypeInfo object, instead of getting them directly from the user.

## 6.6 Can extensions survive new versions of the system?

One of the problems with using extensible systems is that extensions made on one version of the system are often incompatible with a newer version of the system [Mac91]. Extensions written for tailorable systems built using this approach are more likely to survive upgrades to new versions of the system. This is primarily because the extensions do not have unrestricted access to the system. Hence, the new version of the system has only to ensure that it still has the correct open points. Furthermore, should a new version of the system have changes to the original open points — say, changes to the interface of an open point, or deletion of an original open point — the type system will flag the incompatible

extensions when they are loaded into the new version. It will also most likely help the user identify what changes should be made.

The following example illustrates another reason why extensions in this approach are more likely to survive new versions of the tailorable system. Assume that a system has a class `P` which has a virtual method `A` within it. Now assume that the user defines a sub-class `Q` of `P`. In Beta, the user may only further-bind the method `A`, i.e. extend its functionality; the user cannot replace the original `A`. When the user does this, the user *need not* make any assumptions about all the places where `A` may be used. The only assumption the user makes is about the *extension-interface*[15] of `A` in order to determine how it can be extended. On the other hand, if the user were able to replace `A`, the user *would have* to make assumptions about all the places where the original `A` may be used.

Suppose, a new version of the system is released, and in this new version, the `A` is used in a new way. This new use couldn't possibly have been in the set of assumptions made by the replacement user as it didn't exist when the user wrote his/her extension. As a result, installing the old extension into the new version will most likely have unexpected results. Not only this, it will probably be difficult to find the exact cause of the problem. On the other hand, the further-binding user will be better off; as long as the extension-interface of `A` remains the same in the new version, the old extension will work in the new version.

## 6.7 Concluding remarks

An approach for building tailorable systems has been shown. This approach is based on having a generic framework for building systems in the relevant domain, and instantiating it into a specific system that includes an interpreter for the original development language. In addition, the specific system has "open points" through which new classes can be installed. The resulting system is then tailorable in a manner similar to systems which run in residential environments like Lisp and Smalltalk. It supports extensions to the underlying framework at runtime.

The approach has been demonstrated for hypermedia systems. The construction of a highly tailorable hypermedia system has been shown. The tailoring process in order to install a new drawing media-type has also been shown. Problems encountered in saving interpreted objects into a persistent store or OODB have been resolved.

It is important to note that in this approach, the designer of the system has control over where the open points should be, and as a result, has control over what should be

[15] By extension-interface of a class is meant only those aspects of its signature and behavior that affect extensions of it.

tailorable. Thus, the resulting tailorable systems do not allow users to make arbitrary changes to the system, as can be done in Lisp and Smalltalk environments.

An advantage, which is also a limitation, of open-points in Beta is that extensions written by the user cannot easily break functionality already present in the system. This is mainly due to the semantics of Beta, and is discussed in great detail in [Mal94]. In Beta, sub-classes are true extensions of their super-classes; a sub-class refines its super-class, it does not replace parts of its super-class. This limits what can be done by the extension, thus limiting the damage-potential of the extension.

A tailorable system built using this approach will probably *not* be tailorable by a non-programmer user. As has been demonstrated in the paper, the tailoring process requires a small coding effort and a reasonable command of programming. Most use-sites, however, have super-users who could easily be able to do this kind of tailoring [Mac90]. An extension to this approach would be to build a specialized application-oriented language and environment atop these open points; this would make the coding effort much easier. So, the tailorable system, instead of asking the user for Beta source-code files could instead provide an environment for the user to construct this code. This environment could be specialized to, say, the hypermedia domain, and provide constructs to directly support hypermedia-system extensions, e.g. the definition of new media-types.

In this work, the Beta interpreter has been utilized to process and load dynamic extensions. As described in [Mal94], there are other alternatives to this: one could also use the Beta compiler to compile the source and a dynamic linker to link and load the compiled object-code. This alternative approach would, however, not result in a stand-alone system unless the compiler and the linker are embedded within the tailorable system.

To tailor the example hypermedia system with the new drawing media-type, the user must have available, in addition to the system, the interfaces of the framework classes used. In addition, in order for the interpreter to be able to locate the object-code for these framework classes, some symbol-table information, generated during the compilation process of the original system, must also be available. This is all the information about the original system that is necessary. In our experiments, we had available, annotated abstract syntax-trees of the original system. These had more information in them than was really necessary. It should be easy to construct trimmed versions of these trees with only the necessary information. In fact, a prototype implementation of this is available as part of the Mjølner Beta system. So a tailorable system, built using the approach, must be shipped with this additional interface and symbol information in order to be tailorable.

To measure the overhead of using the interpreter approach, the following must be included:

1. the size increase of the hypermedia executable caused by embedding the interpreter.

Initial measurements indicated an increase of 1.3 MB.

2. the size of the linker-generated symbol-table which is part of the hypermedia executable. Ordinarily, this information is not necessary for the hypermedia system to run. However, the embedded interpreter uses this information in order to access object-code in the hypermedia executable; hence it cannot be discarded. This amounts to 2.3 MB.

3. the interface files needed in order to interpret definitions of new media-types. This amounts to 0.2 MB.

The total overhead adds up to 3.8 MB. The size increase caused by embedding the interpreter (1.3 MB) can definitely be further reduced. The value presented is for the first unoptimized version of the interpreter. In addition, forthcoming improvements in the Beta compiler show promise of producing smaller object-code files.

The size increase caused by retaining the linker-generated symbol-table in the hypermedia executable (2.3 MB) can also definitely be further reduced. The Beta compiler produces three symbols for each pattern it compiles, while the interpreter uses only one of these. Hence, it seems like the number of symbols can be reduced by a third; thus, the table should occupy only 0.8 MB. This improvement in the compiler is also forthcoming. Thus, the revised overhead is only 2.3 MB.

In fact, if it were possible to trim this symbol-table to include only those symbols for which there were corresponding definitions in the supplied interface files, the symbol-table overhead could be further reduced. This should be possible by writing a variant of the standard Unix strip command, which could consult the interface files to determine which symbols should be retained, and remove all others.

This overhead of less than 2.3 MB appears small in comparison with that of other source-code-level-tailorable systems. For instance, the hypermedia systems Intermedia and NoteCards discussed in this paper seem to require a much larger overhead to provide similar tailorability. Intermedia was built using MacApp and MPW; major parts of these basic environments are needed, along with the Intermedia code, in order to tailor Intermedia. These core parts of MacApp and MPW in themselves, by rough measurements, add 15 MB in overhead in terms of libraries and executables necessary for tailoring a system like Intermedia. NoteCards was built in the residential Interlisp environment; thus the entire environment is necessary both to run and to tailor NoteCards. An Interlisp image typically adds 12 MB in overhead for a system like NoteCards to be executed and tailored.

This overhead of 2.3 MB can also be compared with that of tailoring the DHM system using the Beta development environment. In this scenario, the user would create the

extensions as shown here, compile them, and relink the application. In order to do the kinds of tailoring we have described, the following would be necessary:

1. the Beta compiler. This amounts to 3 MB.
2. the assembler used by the compiler. This amounts to 0.15 MB.
3. the linker used by the compiler. This amounts to 0.1 MB.
4. the object-code files for the DHM development framework. This amounts to 3 MB.
5. the object-code files for the other basic libraries, e.g. the Xt library. This amounts to at least 1.22 MB.
6. the interface files needed in order to compile the definitions of the new media-types. This is the same as what the interpreter needs and amounts to 0.2 MB.

Thus, the total overhead is 7.67 MB, or 333% of the interpreter-approach overhead.

Yet another scenario can be considered. Here, the compiler would be used to compile the extension, and an incremental — dynamic or static — linker used to link the extension into the DHM executable. In this case, the object-code files would not be necessary (4.22 MB). There would, however, be need for the symbol-table (0.8 MB) as in the interpreter approach. In such a scenario, the overhead would be 7.67 - 4.22 + 0.8 = 4.25 MB, or 184% of the interpreter approach overhead.

Even though the drawing editor, and its presentation class, are interpreted, their performance is acceptable. It, however, remains to be seen how the interpreter would perform in the case of a cpu-intensive media-type like video. The time taken to extend the hypermedia system — with the built-in interpreter — with a new media-type is, however, a small fraction of the time it would take to compile the necessary sources and relink the system.

It has thus been shown that we can provide close to the level of tailorability of systems like NoteCards and Intermedia, but at a fraction of the overhead, and with potentially greater safety.

## Acknowledgements

This work has been supported by the Danish Research Programme for Informatics, grant number 5.26.18.19, and the ESPRIT II/III projects EuroCoOp and EuroCODE. We greatly thank Randy Trigg for his contributions to the development of the DHM development framework. Thanks to Lennert Sloth for helping with the numerous problems

in getting the tailorable hypermedia system working. Thanks to Søren Brandt for help with the persistent store.

# Appendix A

# A brief Beta primer

This brief primer provides a superficial introduction to Beta. Concepts used in the main text, but not presented here, are generally presented where they are used. For more detailed treatment, please consult the Beta book [MMPN93].

A Beta program execution is a collection of objects.[1] 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; a feature that might confuse inexperienced users.

A pattern PP is written as:

```
PP:  P
(#  Decl1; Decl2; ...; Decln                    (* attribute-part *)
  enter Inputs                                     (* enter-part *)
  do Imperatives                                      (* do-part *)
  exit Outputs                                     (* exit-part *)
#)
```

where `P` is the super-pattern. The declarations `Decli` can be of the form:

1. `A: @Q` where `Q` is a pattern. This means that an instance of `Q` is part of every instance of `PP`.

2. `A: ^Q`. This means that a reference to an instance of `Q`, or NONE, is part of every instance of `PP`.

[1] Some ideas for this brief introduction borrowed from [NS90].

3. `QQ: Q (# ... #)`. This is a declaration of a new pattern `QQ` nested within pattern `PP`. It has the same structure as the declaration of `PP`.

The enter-part, `Inputs`, describes the input parameters (formal parameters), the do-part, `Imperatives` describes the actions to be performed (body), and the exit-part, `Outputs` describes the output parameters (return value). The Beta syntax unifies assignments and procedure calls; the symbol `->` denotes both of these operations. For this paper, one should understand `E1 -> E2` as `E2 := E1` if `E2` is an l-value; otherwise, it means `E2(E1)`.

`PP` inherits all the attributes of its super-pattern `P`. The enter-part, do-part, and exit-part of `PP` are derived by combining `P`'s parts with the corresponding parts specified in `PP`.

A pattern may be used as a type or a procedure. As an example of its use as a type consider:

```
Cell: (# Position : (# x : @integer;
                       y : @integer;
                    #)
         loc : @Position;
      #)
```

Here `Position` is used as a type. As an example of a pattern's use as a procedure consider:[2]

```
SpreadSheet :
(#  Cell: (# ... #)
    CreateCell:
    (#  x : @integer;                              (* local variables *)
        y : @integer;
        newCell : ^Cell;
     enter (x,y)
     do new(Cell) -> newCell[];
        x -> newCell.x; y -> newCell.y;
     exit (newCell[])
   #)
   aCell : ^Cell;
do ... (10,10) -> CreateCell -> aCell[]; ...
#)
```

Here `CreateCell` is defined and used as a procedure. In the definition, the construct `new(Cell)` means "create a new instance of pattern `Cell` and return a reference to it."

[2] `new(cell)` is verbose syntax; the actual syntax is `&Cell[]`; the `&` creates an instance, and the `[]` gets a reference to the created instance.

Executing a pattern (e.g. `(10,10) -> CreateCell`) implies creating an instance of it, transferring the input parameters into this instance, executing its do-part imperatives, and returning the output parameters from the instance. We interchangeably use the terms *procedure* or *pattern* for a pattern defined as a procedure; the same applies for patterns defined as functions.

A special variant of a pattern is called a *virtual pattern*. A virtual pattern allows one to defer the specification of some of its details until later. It is declared and specialized as follows:

```
P : (# V :< (# ... #) ... V ... #)
Q : P (# V ::< (# ... #) ... #)
```

In `P`, `V` is declared as a virtual; there is also a reference to `V` within the body of `P`. In `Q`, the inherited virtual `V` is further bound. Hence, in an instance of `P`, the `V`-reference refers to the `V` declared in `P`, while in an instance of `Q`, the same `V`-reference (inherited from `P`) refers to the further binding of `V` in `Q`.

## A.1 Structure Values, and Pattern Variables

In the following program, the pattern variable `X` is declared and initialized first with the structure value denoting the pattern `P2`, and then with the structure value denoting `P3`.[3] The first call to `new(X)` will create a `P2` instance and the second a `P3` instance.

```
P : (#  P1 : (# ... #);
        P2 : P1 (# ... #);
        P3 : P1 (# ... #);
        X : ##P1;                    (* pattern variable declaration *)
     DO
        P2## -> X##;            (* structure value for P2 stored in X *)
        new(X);                              (* use X like a pattern *)
        P3## -> X##;            (* structure value for P3 stored in X *)
        new(X);
     #)
```

[3]The syntax `##P1` implies *reference to pattern* as opposed to *reference to instance of pattern*. In the declaration, `X` is restricted to refer to `P1` or its sub-patterns.
The syntax `P2## ->` implies *get the structure value of the pattern* as opposed to *execute the pattern*.
The syntax `-> X##` implies *store a structure value in the variable* as opposed to *execute the pattern denoted by the variable*.

The structure value for `P2` created by `P2##` encapsulates the prototype for `P2` along with the instance of `P`. Explaining the need for static contexts is beyond the scope of this primer. Refer to [Mad93] for more information on this. Every time an instance is created using the structure value, the saved static context is made the static parent of the instance.

# Bibliography


[ABH+92] P. Andersen, S. Brandt, J. A. Hem, O. L. Madsen, K. J. Møller, and L. Sloth. Workpackage WP5 Task T5.4, Deliverable D5.4: Distributed Object-Oriented Database Interface. Technical Report EuroCoOp deliverable No. ECO-JT-92-3, Jutland Telephone and Aarhus University, 1992.

[AF89] O. Agesen and S. Frølund. Dynamic Link and Load as a Basis for Implementing Extensibility. Technical report, Aarhus University, December 1989.

[AFO89] O. Agesen, S. Frølund, and M. H. Olsen. Persistent and Shared Objects in BETA. Technical Report DAIMI IR-89, Computer Science Department, Aarhus University, Denmark, April 1989. Master's thesis.

[AM87] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324. Springer-Verlag, September 1987.

[App88] Apple Computer, Inc. *HyperCard Script Language Guide: The HyperTalk Language*, 1988.

[App92] Apple Computer, Inc. *On-line version of Applescript User's Guide*, 1992.

[BBE90] G. Bjerknes, T. Bratteteig, and T. Espeseth. Attitudes and knowledge transfer — two important issues of system enhancement. In *Proceedings of the 13th Information Systems Research Seminar in Scandinavia (13th IRIS)*, Turku, Finland, August 1990.

[BDG+89] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation*, 1(3/4):245–394, January 1989. Also X3J13 Document 88-002R, June 1988.

[BDMN73] G. Birwistle, O. Dahl, B. Myrhaug, and K. Nygaard. *Simula Begin*. Auerbach Publishers (New York), 1973.

[Bet89] D. M. Betz. XScheme: An Object-Oriented Scheme. Technical report, P.O. Box 144, Peterborough, NH, 1989.

[BO86] B. G. Bendix and K. Østerbye. Using Active Entities for Debugging and Monitoring of BETA Programs. Technical report, Aarhus University, 1986.

[CM93] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP '93, European Conference on Object-Oriented Programming*, Kaiserslautern, Germany, July 1993.

[DMN68] O. J. Dahl, B. Myrhaug, and K. Nygaard. Simula 67 common base. Technical report, Norwegian Computing Center, Oslo, 1968.

[Dou92] P. Dourish. Computational Reflection and CSCW Design. Technical Report EPC-92-102, Rank Xerox EuroPARC, Cambridge, UK, March 1992.

[Dou93a] P. Dourish. Designing for Change: Reflective Metalevel Architectures for Deep Customisation. Technical Report EPC-93-102, Rank Xerox EuroPARC, Cambridge, UK, January 1993.

[Dou93b] P. Dourish. Towards a Reflective Model of Collaborative Systems. Technical Report EPC-93-114, Rank Xerox EuroPARC, Cambridge, UK, 1993.

[ES90] M. A. Ellis and B. Stroustrup. *C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[Fei82] P. Feiler. A Language-Oriented Interactive Programming Environment Based on Compilation Technology. Technical Report CMU-CS, 82-117, Carnegie Mellon University, Department of Computer Science, 1982.

[FGNR92] G. Fischer, A. Girgensohn, K. Nakakoji, and D. Redmiles. Supporting Software Designers with Integrated Domain-Oriented Design Environments. *IEEE Transactions on Software Engineering*, 18(6):511–522, June 1992.

[Fre93] B. Freeman-Benson. *Personal Communication*, December 1993.

[Fri84] P. Fritzson. *Towards a Distributed Programming Environment Based on Incremental Compilation.* PhD thesis, Dept. of Computer Science, Linköping University, S-581 83 Linköping, Sweden, 1984.

[GHMS94] K. Grønbæk, J. A. Hem, O. L. Madsen, and L. Sloth. Designing Dexter-based Cooperative Hypermedia Systems. *Communications of the ACM*, 37(2), February 1994.

[GHT91] K. Grønbæk, A. Hviid, and R. H. Trigg. ApplBuilder — an Object-Oriented Application Generator Supporting Rapid Prototyping. In *Proceedings of the Fourth International Conference on Software Engineering and its Applications, Toulouse*, December 1991.

[Goo87] D. Goodman. *The Complete HyperCard Handbook*. The MacIntosh Performance Library. Bantam Books, New York, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80. The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[Grø93] K. Grønbæk. Object oriented model for the Distributed Hypermedia Toolkit. Technical Report EuroCODE Workpackage WP2 Task T2.2 report No. CODE-AU-93-10, Aarhus University, Computer Science Department, 1993.

[Gre91] S. Greenberg. Personalizable groupware: Accommodating individual roles and group differences. In L. Bannon, M. Robinson, and K. Schmidt, editors, *Proceedings of CSCW '91, Second European Conference on Computer-Supported Cooperative Work*, pages 17–31, Amsterdam, September 1991.

[GS86] L. N. Garrett and K. E. Smith. Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications*, pages 202–213, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[GT94] K. Grønbæk and R. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, 37(2), February 1994.

[Hed92] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, March 1992. LUTEDX/(TECS-1003)/1-276/(1992).

[Hed93] G. Hedin. *Personal Communication*, March 1993.

[HK91] A. Henderson and M. Kyng. There's No Place Like Home: Continuing Design in Use. In A. Henderson and M. Kyng, editors, *Design at Work*, chapter 11, pages 219–240. Lawrence Erlbaum Associates, 1991.

[HM86] G. Hedin and B. Magnusson. Incremental Execution in a Programming Environment based on Compilation. In *Proceedings of the 19th Hawaii International Conference on System Sciences*, January 1986.

[HM87] G. Hedin and B. Magnusson. Supporting Exploratory Programming in Simula. In *Proceedings of the Fifteenth SIMULA Conference, Jersey*, pages 73–88, September 1987.

[HO90a] W. Harrison and H. Ossher. Extension by Addition: Building Extensible Software. Research Report RC 16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 1990.

[HO90b] W. Harrison and H. Ossher. Subdivided Procedures: A Language Extension Supporting Extensible Programming. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 190–197. IEEE, March 1990.

[HO90c] W. Harrison and H. Ossher. The PlusPlus object environment. Research Report RC 15832, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1990.

[HO93] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA '93, Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[HS94] F. Halasz and M. Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2), February 1994.

[JNZM93] J. A. Johnson, B. A. Nardi, C. L. Zarmer, and J. R. Miller. ACE: Building Interactive Graphical Applications. *Communications of the ACM*, 36(4):41–55, April 1993.

[Joh92] R. E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of OOPSLA '92, Object-Oriented Programming Systems, Languages, and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[JWMM85] K. Jensen, N. Wirth, A. B. Mickel, and J. F. Miner. *Pascal User Manual and Report: Revised for the ISO Pascal Standard.* Springer-Verlag, 1985.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[Kee89] S. E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley Publishing Company, 1989.

[Kic92] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of IMSA'92 (Workshop on Reflection and Meta-level Architectures)*, Tokyo, Japan, November 1992.

[Kic93] G. Kiczales. Traces (A Cut at the "Make Isn't Generic" Problem). In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software*, LNCS 742, Kanazawa, Japan, November 1993. Springer-Verlag.

[KL92] G. Kiczales and J. Lamping. Issues in the Design and Specification of Class Libraries. In *Proceedings of OOPSLA '92, Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[KMMPN86] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Dynamic Exchange of Beta Sytems. Draft, March 1986.

[KMMPN87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA programming language. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*. MIT Press, 1987.

[KN93] M. Kyng and P. A. Nielsen. Domain Modeling and Application Frameworks. In *Proceedings of 16th IRIS — Information systems Research seminar In Scandinavia*, Copenhagen, Denmark, August 1993. University of Copenhagen, Department of Computer Science.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[Lam93] J. Lamping. Typing the specialization interface. In *Proceedings of OOPSLA '93, Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–214, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[Lie87] H. Lieberman. Reversible Object-Oriented Interpreters. In *Proceedings of ECOOP '87, European Conference on Object-Oriented Programming*, Paris, France, June 1987. Springer Verlag. LNCS 276.

[LM84] M. Løfgren and B. Magnusson. An Execution Environment for Simula on a Small Computer. Technical Report LUTFD2/(TFCS-3006)/1-96/1984, Department of Computer Science, Lund University, Sweden, March 1984.

[Lun92] Lund Software House AB, Box 7056, S-220 07 Lund, Sweden. *SIMULA User's Guide*, 1992.

[Mac90] W. Mackay. Patterns of Sharing Customizable Software. In *Proceedings of CSCW '90*, Los Angeles, CA, October 1990. ACM Press.

[Mac91] W. E. Mackay. Triggers and Barriers to Customizing Software. In *Proceedings of CHI '91*, pages 153–160. ACM Press and Addison Wesley, April 1991.

[Mad93] O. L. Madsen. The Implementation of BETA. In *Object-Oriented Environments: The Mjølner Approach*. Prentice Hall International, 1993. ISBN 0-13-009291-6.

[Mag93] B. Magnusson. The Mjølner Orm system. In *Object-Oriented Software Development Environments*. Prentice Hall International, 1993. ISBN 0-13-009291-6.

[Mal93a] J. Malhotra. Dynamic Extensibility in a Statically-compiled Object-oriented Language. In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, LNCS 742, pages 297–314, Kanazawa, Japan, November 1993. Springer-Verlag.

[Mal93b] J. Malhotra. Extensibility as the basis for Incremental Application Generation. Technical report, Computer Science Dept., Aarhus University, November 1993.

[Mal93c] J. Malhotra. On the Implementation of an Interpreter for Building Extensible Applications. Technical report, Computer Science Dept., Aarhus University, October 1993.

[Mal94] J. Malhotra. On the Construction of Extensible Systems. In *Proceedings of TOOLS EUROPE '94 (to appear)*, Versailles, France, March 1994.

[MC93] P. Mulet and P. Cointe. Definiton of a Reflective Kernel for a Prototype-Based Language. In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, LNCS 742, pages 128–144, Kanazawa, Japan, November 1993. Springer-Verlag.

[MCLM90] A. MacLean, K. Carter, L. Lövstrand, and T. Moran. User-Tailorable Systems: Pressing the Issue with Buttons. In *Proceedings of CHI'90*, pages 175–182, Seattle, Washington, April 1990.

[Mey86] N. Meyrowitz. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications*, pages 186–201, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988. Series in Computer Science. ISBN 0-13-629031-0.

[MHRJ91] C. C. Marshall, F. G. Halasz, R. A. Rogers, and W. C. Janssen Jr. Aquanet: a hypertext tool to hold your knowledge in place. In *Proceedings of Hypertext '91*, pages 261–275, San Antonio, Texas, December 1991. ACM.

[MI92a] Mjølner Informatics. *The Mjølner BETA System — The BETA Source-level Debugger — User's Guide*, 1992. MIA 92-12.

[MI92b] Mjølner Informatics. *The Mjølner BETA System, Reference Manual*, 1992. MIA 90-04(1.0).

[MI93] Mjølner Informatics. *Sif, A Hyper Structure Editor, User's Guide*, 1993. MIA 91-11(1.0).

[Mic91] Microsoft, Inc. *Microsoft Word Version 5.0 User's Guide*, 1991.

[Mic93] Microsoft, Inc. *Microsoft Excel Version 4.0 User's Guide*, 1993.

[MMMP90] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. In *Proceedings of OOPSLA/ECOOP '90*, pages 140–149, October 1990. Published as SIGPLAN Notices, Volume 25, Number 10.

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley Publishing Company, 1993. ISBN 0-201-62430-3.

[Mør93] A. Mørch. Designing for Tailorability: Coupling Artifact and Rationale. In *Proceedings of the 16th Information Systems Research Seminar in Scandinavia (16th IRIS)*, Copenhagen, Denmark, August 1993.

[MT81] L. Masinter and W. Teitelman. The Interlisp Programming Environment. *Computer*, 14(4):25–34, April 1981.

[MY89] A. MacLean and R. M. Young. Design Rationale: The Argument Behind the Artifact. In *Proceedings of CHI'89*, pages 247–252, May 1989.

[NG87] D. Notkin and W. G. Griswold. Enhancement through Extension: The Extension Interpreter. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minn., June 1987.

[NGT92] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[Nør92] K. Nørmark. From Hooks to Open Points and back again. Technical report, Aalborg University, March 1992.

[NS90] C. Nørgaard and E. Sandvad. Reusability and Tailorability in the Mjølner BETA System. Technical Report DAIMI PB-300, Aarhus University, January 1990.

[Nyg89] K. Nygaard. Profession Oriented Languages. In P. B. Andersen and T. Bratteteig, editors, *Computers and Languages at Work, SYDPOL Working Group 2*, pages 19–27. Institute of Informatics, Oslo University, 1989. Research Report No. 126.

[OH92] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Proceedings of OOPSLA '92, Object-Oriented Programming Systems, Languages, and Applications*, pages 25–40, Vancouver, October 1992. ACM. Published as ACM SIGPLAN Notices, volume 27, number 10.

[Oss87] H. L. Ossher. A Mechanism for Specifying the Structure of Large, Layered Systems. In *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, 1987.

[PSS87] C. H. Pedersen, L. B. Sørensen, and P. F. Sørensen. The BETA shadow language — a basis for high level debugging. Technical Report DAIMI IR-77, Aarhus University, November 1987.

[Rao91] R. Rao. Implementational Reflection in Silica. In P. America, editor, *Proceedings of ECOOP '91, European Conference on Object-Oriented Programming*, LNCS 512, pages 251–267, Geneva, Switzerland, July 1991. Springer-Verlag.

[Rya90] B. Ryan. Scripts Unbounded. *BYTE Magazine*, pages 235–240, August 1990.

[Sch86] K. J. Schmucker. Introduction to MacAPP. In *Object-Oriented Programming for the MacIntosh*, chapter 4, pages 83–129. Hayden Book Company, Hasbrouck Heights, New Jersey, 1986.

[ST91] L. Suchman and R. Trigg. Understanding Practice: Video as a Medium for Reflection and Design. In A. Henderson and M. Kyng, editors, *Design at Work*, chapter 4, pages 65–89. Lawrence Erlbaum Associates, 1991.

[Sta84] R. M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In *Interactive Programming Environments*. McGraw-Hill, 1984.

[Sta85] R. Stallman. *GNU Emacs Manual*. Free Software Foundation, 1985.

[Ste90] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[Szy92] C. A. Szyperski. Write-Ing Applications: Design of an Extensible Text Editor as an Application Framework. In G. Heeg, B. Magnusson, and B. Meyer, editors, *Proceedings of TOOLS '92*, pages 247–261, Dortmund, Germany, 1992. Prentice Hall.

[TMH87] R. H. Trigg, T. P. Moran, and F. G. Halasz. Adaptability and Tailorability in Notecards. In *INTERACT 87, Stuttgart, Germany*, September 1987.

[Tri93] R. Trigg. Participatory Design meets the MOP: Informing the design of tailorable computer systems. *Submitted to ACM Transactions on Information Systems*, 1993. Previously appeared as: "Participatory Design meets the MOP: Accountability in the design of tailorable computer systems," Proceedings of the 15th IRIS (Information systems Research seminar In Scandinavia), G. Bjerknes, T. Bratteteig, K. Kautz (eds.), August 1992, Larkollen, Norway. pp 643-656.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87, Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Orlando, October 1987. ACM. Published as ACM SIGPLAN Notices, volume 22, number 12.

[vd91] G. van Rossum and J. de Boer. Interactively Testing Remote Servers Using the Python Programming Language. *CWI Quarterly*, 4(4):283–303, December 1991.

[Yok93] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, LNCS 742, pages 145–162, Kanazawa, Japan, November 1993. Springer-Verlag.

[ZC92] C. L. Zarmer and C. Chew. Frameworks for Interactive, Extensible, Information-Intensive Applications. In *Proceedings of UIST'92*, pages 33–41, November 1992.