Ph. D. Dissertation
July, 1996

ZYPHER

Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia

Serge Demeyer

Vrije Universiteit Brussel
Departement Informatica

"*And since dissertations can be written about everything under the sun, the number of topics is infinite. Sheets of paper covered up with words pile up in archives sadder than cemeteries, because no one ever visits them, not even on All Souls' Day. Culture is perishing in overproduction, in an avalanche of words, in the madness of quantity. That's why one banned book in your former country means infinitely more than the billions of words spewed out by our universities.*"

Milan Kundera, The Unbearable Lightness of Being
(Part Three: Words Misunderstood — Sabina's Country)

Table of Contents

Abstract

The dissertation concerns a study of state of the art object-oriented software engineering applied within the domain of open hypermedia systems. The results of this study are discussed within the context of a software artefact named Zypher.

The scientific contribution of this work is situated in the domain of object-oriented software engineering. The contribution is a proper combination of frameworks and meta-object protocols, which are two promising techniques in object-oriented software engineering. We show that, when combining both approaches, explicit representations of framework contracts are part of a meta-object protocol. This insight is valuable in the design of meta-object protocols.

The design of meta-object protocols is crucial in the development of large scale software systems, because it leads to greater flexibility. Flexibility is a key requirement, since large scale software systems are bound to satisfy ever changing demands, hence must be adapted continuously. This flexibility requirement is very prominent in open hypermedia, the research domain where the results are validated experimentally.

The archetype for the kind of software envisioned is an open hypermedia system supporting a group of engineers in a design and manufacturing process. Such a system must handle the management of several tens of gigabytes of data stored in different devices, contained in millions of individual design elements maintained in various formats, organised in densely interconnected graphs representing numerous relationships and —probably the most difficult requirement— constantly evolving while the design and manufacturing process proceeds. Without a doubt, flexibility is a key requirement in these engineering support systems.

We propose tailorability as a means to cope with the flexibility requirement, and we identify at least three levels of tailorability to handle different kinds of flexibility. Domain level tailorability is needed to adapt a software system to changing needs, such as the incorporation of new data formats. System level tailorability copes with requests about the system performance, data security or information integrity. Configuration level tailorability handles demands such as cross-platform portability and run-time reconfiguration.

We show that object-oriented software engineering techniques are valuable for the development of tailorable systems. More precisely, domain level tailorability can be achieved by building an object-oriented framework. Extending the framework with a meta-object protocol attains system level tailorability and yet another meta-level implements configuration level tailorability.

Acknowledgements

I thank my promotor Prof. Theo D'Hondt. He pushed me to define this meta-object protocol in my Zypher framework and I confess that I doubted whether there would be one. Like all good promotors, he was right and I was wrong. I am forever in depth with Patrick Steyaert. He was my mentor during the painful days and tempered my assaults until they fitted the thesis format. Patrick also advised me to form a circle of people that could help me in filtering ideas: Koen De Hondt, Roel Wuyts, Wim Codenie and Wolfgang De Meuter, thanks for helping me out. Valuable, among others in proof-reading were Carine Lucas, Kim Mens and Marc Van Limbergen. Karel Driesen wasn't around when I finished the text, but I still owe him a lot for all the lively discussions after a pint of beer and a borrowed cigarette. To all the other people at the Programming Technology Lab, the Computer Sciences Department and the University that I worked with during these years, thanks for being nice colleagues. I would like to mention Viviane Jonckers for being an outsider with valuable comments, and Eddie Vandijck for being a believer.

Using internet I was able to manage some international contacts as well. Thanks to Uffe Kock Wiil for inviting me to organise the second open hypermedia workshop; it was a great occasion to enter the open hypermedia community. Thanks to Kasper Østerbye (and Uffe) for sending me drafts of what was to become the "Flag Taxonomy on Open Hypermedia Systems" and the helpful discussions over e-mail afterwards. Helen Ashman provided me with a draft of a paper discussing requirements lists for open hypermedia systems, which was a source of inspiration. It was good to know that I was not the only one combining design patterns with hypermedia, Alejandra Garrido does parallel work. Eric Steegmans forced me to reconsider the white-box versus black-box debate, which provided me with some valuable new insights.

I got into the hypermedia field from a computer supported co-operative work angle. I am especially grateful to Michel Tilman, for pointing me in the Hypermedia domain. I would like to thank Dirk Kenis for running the DSide project with all its ups and downs and for helping me with the apposition of this thesis. I am grateful to Wim Lybaert for working on a combination of Smalltalk and HTML/HTTP to make a version of DSide that was accessible via the web. Niels Boyen and Thierry Marchant helped during the development of other DSide prototypes.

The ARGO was an environment that confronted me with the problem how object-oriented ideas survive in real projects; to all the people at the AIV (especially Martine Devos), thanks for keeping me out of the ivory tower. Some people at OOPartners where helpful for both material and spiritual support, thanks to Wilfried "Wiffel" Verachtert, Arlette Vercammen and Peter Oyserman.

I thank my parents as well. First of all for giving me the opportunity to study, it was the first step towards this thesis. Also for all the love and support during these 30 years. My brothers, the rest of my family and family in law provided for the necessary distraction once in a while. Special thanks go to my brothers for the nice artwork in this dissertation.

And finally I would like to thank my wife, Ann. I know she doesn't believe me, but I wouldn't have finished this thing without her.

Serge Demeyer

July, 1996

PART *1*

The Zypher Experiment

Introduction

Nowadays, a considerable amount of effort is spent on the design of open systems. This phenomenon can be observed in operating systems (e.g. UNIX and OS/2), databases (e.g. Exodus and CORBA), inter application communication (e.g. OLE and OpenDoc), tailorable software (e.g. Emacs and AutoCAD), programming languages (Smalltalk and CLOS) and last but not least —certainly with the advent of the world-wide web— hypermedia.

A closer examination of these fields reveals that there are a number of common themes underlying all these efforts. A first one is *interoperability*, an idea that starts from the perception that software systems are getting bigger and bigger, but rarely incorporate new ideas. That is, in most occasions each major release clones features from other complementary systems. Rather than ending up with a huge system that behaves like a common denominator of many other systems, one wants users to choose their favourite and specialised systems and provide some form of co-operation facilities between them. A second theme in the design of open systems is *extensibility*, where system designers acknowledge that a single application will never be able to serve all the needs of all of its users. Rather than building static systems, system designers want to provide some form of extensibility so that end users can extend the systems to their needs. A final theme is *distribution*, where people want to share resources by connecting workstations and peripheral hardware through computer networks.

From an end users perspective the interest in these themes is justified, since they try to provide quite simple solutions for quite simple problems. However, when software designers must incorporate these solutions into their systems, they are faced with a number of complex issues and questions. Each software designer comes up with different answers and this again gives rise to the enormous pool of standards, languages and protocols we find in today's software systems. To cope with such a chaotic diversity systems get bigger and bigger, so consume more resources, contain more bugs and are more expensive to purchase. Something needs to be done to stop this evolution and this explains the motivation for the numerous efforts on designing open systems: they are widely recognised as an attempt to address this problem.

This Ph.D. research is a contribution to the same attempt. As Ph.D. research is well suited for exploring the more risky aspects of open systems, we have examined state of the art techniques in other domains, applied these techniques to the design of a particular open system situated in the hypermedia domain, identified combinations that work, clarified why these combinations work and tried to generalise the results afterwards.

The techniques we applied in this work stem from the domain of software engineering. In this domain it is recognised (see among others the influential article "No Silver Bullet: Essence and Accidents in Software Engineering" [Brooks'87]) that software systems are more difficult to build than any other kind of systems built in any other engineering discipline. This is motivated from the observation that software systems are intrinsically complex, must deal with the non-conformity of numerous human institutions and systems, is subject to a constant demand for change and is inherently invisible thus hard to conceive. One of the ways to deal with the inherent difficulties of constructing software systems is called *incremental development*. In natural systems like the human eye or brain, all complexity is the result of evolution, so proponents of incremental development feel that complexity in software systems should be the result of evolution as well. So in the incremental development approach, a software system is regarded as a flexible construction that is constantly adapted, rather than a static device that can not be changed once it has left the design table.

The software engineering community puts forward various approaches towards incremental development. Two of them —*object-oriented frameworks* and *meta-level abstraction*— have proven their value in the development of commercially available systems. The former is a design specification technique for reusable designs in a particular problem domain. The latter is an abstraction technique that provides specific interfaces to manage system mutations.

From these short descriptions, it follows that both techniques have different goals — frameworks aim at design reuse while meta-level abstractions aim at adaptability. Yet, both techniques are applicable in an incremental development approach — frameworks because the same design is likely to be reused in future instances of a system, meta-level abstractions because they facilitate system mutations. So we may conclude that both techniques are *complementary* with respect to the incremental development approach.

Returning to the main theme of the dissertation, it seems that the incremental development approach is quite suitable when constructing open systems. Open systems address the chaotic diversity that follows from the numerous standards, languages and protocols that exist in today's software systems. Natural systems cope with chaos by evolution, so incremental development is likely to be applicable to the design of open systems. This brings us to the first research hypothesis formulated below.

Research Hypothesis 1

> *Object-oriented frameworks and meta-level abstraction are two complementary techniques in the design of open systems.*

This hypothesis embodies the main ideas that have been driving our research. On the one hand, we want to understand how complementary both techniques are and —based on this insight— propose a design methodology as a proper combination of object-oriented frameworks and meta-level abstraction. On the other hand, we want this design methodology to address problems and issues that are encountered in the design of real systems.

In the domain of software engineering, it is common practice to validate techniques experimentally [NationalAcademy'94] — researchers build prototype systems for a particular application domain to investigate benefits and drawbacks of a technique, and generalise these results to a broader set of application domains afterwards. Such a prototype system is called a software artefact and in our dissertation this is role is played by the Zypher system.

In such an experimental validation, the choice of the target application domain is of prime importance: the software artefact's application domain must be *representative* for a larger set of application domains. The idea is that the prototype system should be small enough so that it can be built and adapted rapidly, yet scalable for the kind of problems encountered in real systems. When the software artefact's application domain is representative for a larger set of application domains, the results obtained in the specific software artefact can be generalised to the larger set of applications.

We have chosen our prototype system to be an open hypermedia system. Open hypermedia is about the management of large, complex and heterogeneous information structures and as such, open hypermedia covers the common themes underlying all open systems — interoperability, extensibility and distribution. So it is likely that the results of an open hypermedia software artefact can be generalised to the larger set of open systems, which brings us to the second research hypotheses formulated below.

Research Hypothesis 2

> *Open hypermedia systems are representative for the larger set of open systems.*

The generic set of open systems is quite large. Besides open hypermedia systems, it comprises domains like operating systems, database systems, inter application communication standards, tailorable software systems and programming languages. Yet we chose hypermedia as the representative element and we had a very special motivation to do so.

The archetype for the kind of information to be managed by an open hypermedia system is the engineering documentation used in the design and manufacturing process of airplanes. Such a system must handle the management of several tens of gigabytes of data stored in different devices, contained in millions of individual design elements maintained in various formats, organised in densely interconnected graphs representing numerous relationships and —probably the most difficult requirement— constantly evolving while the design and manufacturing process proceeds.

Open hypermedia is recognised as a *technological cornerstone* for systems supporting airplane manufacturing. That is, it is possible to construct such a system without incorporating open hypermedia, but neglecting the technology implies the loss of valuable concepts and strategic advantages.

If open hypermedia is recognised as an invaluable ingredient in airplane manufacturing, it may support other engineering disciplines as well. In the context sketched above, a logical candidate is the software engineering discipline, which brings us to the third research hypothesis formulated below.

Research Hypothesis 3

Open hypermedia is a technological cornerstone for supporting software engineering.

This dissertation is concerned with the examination of the three research hypotheses. We will devise a model for open hypermedia systems that can be generalised into a model for open systems; we will propose a design methodology based on a proper combination of object-oriented frameworks and meta-level abstraction and we will experiment with the notion of a framework browser as a tool that supports the management of object-oriented frameworks.

Road Map

The dissertation is divided in three parts. The first part presents an overview of the important research fields where this work found its roots. It starts with an overview of open hypermedia, including a generic model for open hypermedia systems and a scenario for a hypothetical framework browser. The former is based on the notion of a design space with three levels of tailorability. This part of the dissertation supports the second research hypothesis and introduces some arguments for the third hypothesis (see "The Zypher Perspective" - p.30). Next follows a survey of the two investigated techniques (object-oriented frameworks and meta-object protocols) followed by a discussion on how those techniques can be used to build an open system — that is a system that adheres to the principles argued in the design space with three levels of tailorability. This part of the dissertation introduces the main arguments for the first research hypothesis (see "The Zypher Contribution" -p.63).

The second part describes the software artefact used in our experiments, i.e. it provides a description of the Zypher framework in design pattern form. In this part, each element in the design and its contribution to the overall framework is explained. The design patterns are written to stand on themselves, so many of the ideas and argued in the first part will be repeated here (i.e. the design space, the meta-object protocols, …). The second part serves two purposes. Besides being a description of how we applied object-oriented frameworks and meta-object protocols in the design of the Zypher framework (i.e. the experimental validation of the first research hypothesis), it also reports on how we build a prototype of a framework browser (i.e. the experimental validation of the third research hypothesis).

Finally, the last part includes the conclusion and appendices. The conclusion reconsiders the three research hypotheses in the light of all that precedes and review the results from a broader perspective. The appendices includes the apposition —which is an necessary part of a Ph.D. dissertation at our university— and readers aids like a reference list, an index, a list of figures and a table of contents.

While the first part is quite conventional, the idea of non-linearity (hypermedia remember) is crucially important in the second part, i.e. the design pattern description of the Zypher framework. Indeed, the real power of design patterns lies in the idea of overlapping many of them in a single space. Overlapping patterns include several mutual references, hence the importance of non-linearity. *Nevertheless we want to stress that for a first tour, the sequential path is probably the best. Only for an in depth study, the referencing scheme is essential to understand how all patterns work together to achieve the overall design.*

Of course this text is available in hypertext format as well. Check the Zypher home page *http://progwww.vub.ac.be/prog/pools/Zypher/zypher.html*, to read the most recent version and other material created as part of the Zypher project.

CHAPTER *1*

Open Hypermedia

Not so many years ago, it would have taken quite a long time to explain to people the basic notions of hypertext (or hypermedia — both names are used interchangeably throughout this text). Oh yes, system administrators would have used technical documentation in hypertext form and some Macintosh users might even had experiences with HyperCard stackware. Still, most people would have shown big question marks hearing definitions like "hypermedia is a style of building systems for information representation and management around a network of multi-media nodes connected together by typed links" [Halasz'87] and certainly would not see the tremendous possibilities of what has been called the navigational paradigm to computing.

Today, this is different. With the coming of the world-wide web, almost all computer users are well aware of basic hypermedia concepts like links, anchors, nodes, non-linearity and navigation. The establishment of cyber cafes takes away the obstacle of not owning a computer: people can visit specially designed places to surf the web. In the United States, and more and more in Europe, the URL's are an integral part of advertising and hypermedia is even becoming visible in the streets now.

If hypermedia has finally hit the mass market, then why is it still appropriate to write a Ph.D. dissertation on such a topic ? Well, as you will notice reading this text, hypertext and hypermedia research has a long standing history of fruitful research and the computer science community can —and will— pluck more of those inspiring ideas in the time to come. One of this ideas is open hypermedia, and that is the topic discussed in this chapter.

Hypertext & Hypermedia

With the explosive growth of the world-wide web, people are generally acquainted with the basic hypermedia concepts. Still, there is more to hypermedia than there is to the world-wide web and this section introduces many of the important ideas living in the hypermedia community.

There are several ways to give an overview of those ideas. One could try for an analytical approach by partitioning the space of hypermedia systems along a number of dimensions. This approach has been taken in [Halasz'87], [Halasz'91] and also in [Akscyn,McCracken,Yoder'87], [Akscyn'91].

The problem with an analytical approach is that it is hard to illustrate why a certain dimension is 'good' way to partition the space. To avoid this problem, we chose to provide a definition and basic vocabulary of hypermedia, followed by brief historical overview that introduces the reader to the major milestones in hypermedia research. As a rough guide we classified the milestones in generation of hypermedia systems. As with all brief overviews, many contributions of many interesting projects could not be included, so be aware that this is by no means the ultimate overview of hypermedia research.

What is Hypermedia ?

Hypertext, Non-linearity and the Navigational Paradigm

The simplest way to define hypertext is to contrast it with the more traditional approaches to text. Traditional information sources like books are based on the idea of a linear processing of information: readers are supposed to read from the first page to the last page. Through the history of paper publication, people have found ways to circumvent this linear processing of ideas (i.e. a phone book, a dictionary, a table of contents, an index, …) and the invention of desktop computers opened a new range of promising solutions.

Hypertext is the name of the research domain that investigates the possibilities of manipulating non-linear text: the idea is that computers can be used to help users create, maintain and navigate between chunks of text to form structures and shape information. Navigation is the basic operation that brings information explorers from one chunk to another, which explains why hypertext is sometimes called the navigational paradigm.

Note that, although hypertext contrasts with traditional text approaches, a lot of the traditional vocabulary of sequential text remains appropriate. The hypertext community adopts terms like document, reading, reader, authoring, author and viewer instead of the more computer oriented ones like file, using, user, implementing, implementor and application.

Hypermedia = Hypertext + Multi-media

The idea of hypertext is quite old, but with the advent of powerful workstation technology, it became feasible to extend the notion of hypertext beyond mere text-processing. Instead of manipulating chunks of text, one could also shape information structures containing pictures, video, sound, …. . Hypermedia —a contraction of the words Hypertext and Multi-media— is a name invented to stress this change of emphasis. However, like most material in the field, we use both terms interchangeably.

Node, Link and Anchor

Many hypermedia projects have attempted to build models to support this navigational paradigm. Each project invented its own terminology resulting in an enormous amount of words referring to similar concepts. Still, the hypermedia community has managed to reach consensus on some basic vocabulary. As the first hypermedia projects started from a data model based on graphs, the basic vocabulary inherits quite a lot from graph theory.

The basic unit of information in a hypermedia system is called a node, which may contain arbitrary information like text, graphics, video, sound, … . Some hypermedia systems allow a special kind of node, called a composite node, which can contain other nodes.

A hypermedia system organises nodes into navigational structures. A link is the name for an explicit representation of a navigation relationship that exists between nodes. The navigation effect —essential to all hypermedia systems— is achieved by traversing the links. Links can be binary or n-ary; the latter allows an arbitrary number of endpoints for a link, the former allows exact two. Most hypermedia systems enforce directed links, which can be traversed in one direction only. The traversal direction of the link determines a source (the starting point of the navigation operation) and a destination (the arrival point). Many hypermedia systems allow typed links, which have some kind of label attached to them so that the reader may infer the purpose of the link without actually traversing it.

Links can be connected to whole nodes, but in most occasions links have special purpose connection points called anchors. An anchor is thus an explicit representation of a referencable substructure of a node serving as an endpoint of a link.

Hypertext & Hypermedia Definition

In this dissertation, we adopt the definition of the influential "Seven Issues: revisited" keynote address [Halasz'91].

"Hypermedia is a style of building systems for the creation, manipulation, presentation and representation of information in which:
- *the information is stored in a collection of multi-media nodes*
- *the nodes are explicitly or implicitly organised into one or more structures (commonly, a network of nodes connected by links)*
- *users can access information by navigating over or through the available information structures."*

There are two important remarks to make on this definition. First of all, it is a revised version of the definition that appeared in [Halasz'87]. The original version reads as follows: "Hypermedia is a style of building systems for information representation and management around a network of multi-media nodes connected together by typed links". In the revised version, Halasz explicitly recognises that the means to construct navigational structures (i.e. nodes and links) are less important than the idea of navigation itself. This change of emphasis was driven by promising new approaches for structuring navigation relationships (i.e. spatial hypertexts).

The second remark is that one cannot underestimate the importance of the word "style" in this definition. To quote Jacob Nielsen [Nielsen'90] - p.02 "*Many non-hypertext computer techniques may at least match various aspects of the definition of hypertext, but true hypertext should also make users feel that they can move freely through the information according to their own needs. […] When asked whether I would view a* *certain system as hypertext, I would not rely so much on its specific features, commands, or data structures, but more on its user interface 'look and feel'.*"

On this matter of style the last word is not said. Perhaps the closest specification of the essence of hypertext style is the one from Helen Ashman [Ashman'94]: "*The common thread that runs through most definitions is the idea that hypermedia permits users to customise the presentation of the information in some way. At least some part of the information presentation is in the user's control. In the simplest cases, only the order of traversal is decided by the user. But the more complex and powerful hypermedia systems let the user add to the hypermedia link structures as well. In every case, the user creates a personalised environment tailored to meet their needs. It is this personalisation and customisability of the information presentation that characterises hypermedia. This incorporates the idea of non linearity, since non linearity gives control of the order of traversal to the user*".

The style aspect in the definition is a subjective criterion and makes it very hard to judge whether a system is hypertext and hypermedia. Nevertheless, or maybe just as a consequence, the lack of a precise definition has not tempered the hypermedia community to provide numerous fruitful ideas and we discuss the most important ones below.

A Guided Tour of Hypermedia

The earlier periods of this guided tour are primarily based on the book "Hypertext & Hypermedia" by Jacob Nielsen [Nielsen'90]. Other inspiration was drawn from the annual conferences on Hypertext: the first two organised in the US [HT'87], [HT'89] and from then on alternating between Europe and the US [ECHT'90], [HT'91], [ECHT'92], [HT'93], [ECHT'94], [HT'96]. The keynote address by Randy Trigg [Trigg'96] served as a backbone for the classification in generations.

a) The Pioneers — Memex, Augment and Xanadu

Memex — Vanevar Bush (1945)

Most people agree that Vanevar Bush is to be considered the founding father of hypermedia research. His paper "As we may think" [Bush'45], described an imaginary "Memex" system based on microfilm technology (!) to support scientists in their task of tracing the enormous amount of information that was published in their discipline. The idea was that people could use a machine to link together fragments of different documents and pass this annotated material to colleagues: the very same idea that started of the world-wide web project [Berners-LeeEtAl'94].

Augment/NLS — Douglas Engelbart (1962-1976)

While the Memex system coined the idea of using a machine to maintain associative links between document fragments, it was the Augment project that —in a time where computers were used mainly as number crunchers and instructed through punch cards— laid the basis for computer technology as tools for realising a "Conceptual Framework for Augmenting Human Intellect" [Engelbart'95]. This project introduced a second major theme of the future hypermedia research, that of computer supported co-operative work (CSCW). Besides that, the Augment project invented nearly half of the concepts of today's desktop environments, such as e-mail, teleconferencing, windows, the mouse pointing device, etc.

The hypermedia research community still acknowledges the great influence of the vision of Douglas Engelbart and has established a yearly prize in honour of his person.

Xanadu — Theodor Nelson (1965-…)

It was the Xanadu project that actually coined the words hypertext and hypermedia in a vision that all information would be published in a universal information repository, accessible via computers ("literary machines"). Although the Xanadu Operating

Company (see http://www.mpx.com.au/) is entirely devoted to the implementation of that vision, it was the world-wide web project [Berners-LeeEtAl'94] that came nearest to a such a universal information space. Nevertheless, the Xanadu project will always be credited as the first project that performed constructive thinking on problems like universal referencing schemes [Nelson'87], copyright issues, versioning, etc.

The three pioneering hypermedia systems share the idea of using machines to link together fragments of information. Yet, they came to this idea from a completely different angle. Memex envisioned a system supporting scientists in organising information, so the emphasis was on authoring associative links and on the annotation of material. Augment/NLS was a prototype of an office information system, where co-operating people work on shared material, hence the need for fine grained concurrency control mechanisms, integration of tools and user-interface. Xanadu aimed for very large and distributed information systems, so there the emphasis was more on coarse grained concurrency control algorithms, replication of information and referencing schemes.

b) The Monolithic Hypermedia Systems — NoteCards and Intermedia

After these three pioneering projects, lots of activity went on in hypermedia community. We refer the interested reader to [Conklin'87] for an overview and a comparative study of those earlier systems (with names like Boxer, CREF, Emacs INFOR, IBIS, Intermedia, KMS, Neptune, NoteCards, PlaneText, Document Examiner, SYNVIEW, Textnet, Hyperties, WE — and the list is not yet complete) and concentrate on two of them: NoteCards and Intermedia. Our choice was guided by the significant impact of both systems on all later hypermedia research. Based on the classification of [Trigg'96], we see them as representative for the generation of monolithic hypermedia systems.

NoteCards

NoteCards [Halasz'87] was developed as a research prototype at Xerox Parc within the InterLisp environment. Programmers were able to extend the system by means of an API of over a 100 routines so that the NoteCards system could be tailored to their needs. The NoteCards prototype has been used outside Xerox Parc in numerous settings and as such the system had quite a lot of impact on the hypermedia community.

The design of the system was based on a 'notecard': an electronic generalisation of the 3x5 paper notecard that could contain an arbitrary amount of some editable information (text, graphics, …). Cards could be linked together by means of typed directional links; users could follow links by clicking on the source of a link to pop up the target card. The 'browser' allowed users to edit a network of interlinked notecards and the 'filebox' enabled the collection of notecards into composites.

Hypermedia Research Agendas

The main reason why the NoteCards system could not be omitted from this overview is the fact that it was used twice as a foil against which to explore some of the major limitations of two generations of hypermedia systems and as such caused the definition of two agendas still driving the research in the hypermedia community. The first (see [Halasz'87]) put forward a list containing the items: (1) search and query, (2) composite nodes, (3) virtual structures, (4) computational engines, (5) versioning, (6) collaborative work and (7) tailorability. This agenda was revised in 1991 (see [Halasz'91]) in the list: (1) ending the tyranny of the link, (2) open systems, (3) support for collaborative work, (4) user interfaces for large information spaces, (5) very large hypertexts, (6) tailorability and extensibility, (7) computation in (over) hypermedia networks, (8) defining the hypermedia markets, (9) standards, (10) publishing hypertexts. Figure 1 shows how both research agendas map onto each other. It would take us too long to study each of the items in detail, but some of them are covered in the section open hypermedia (p.22).

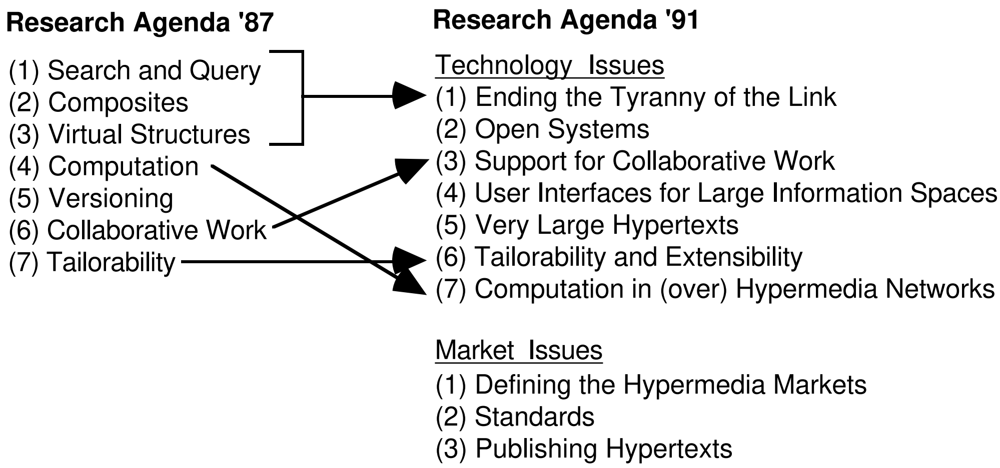


Figure 1: Hypermedia Research Agendas

Intermedia

Intermedia ([Yankelovich,Meyrowitz,VanDam'85], [Meyrowitz'86], [Catlin,Bush,Yankelovich'89], [HaanEtAl'92]) is another hypermedia system that has influenced —directly or indirectly— all subsequent hypermedia research. Unfortunately, the Intermedia system was only available on the AUX platform (a UNIX operating system for the Macintosh), which has severely hampered it to spread into real applications.

Intermedia was designed to demonstrate operating system developers that hypermedia functionality should be integrated into the desktop computing environment and was the first advocate for an open hypermedia system philosophy [Meyrowitz'89]. The technology was not available at that time, which explains why Intermedia is still to be considered a monolithic system.

Linking Protocol

In Intermedia, nodes are called 'documents' to be edited by editors part of the Intermedia system. The Intermedia editors represent standard desktop applications like word processors, drawing packages, e-mail, …. The Intermedia project coined the idea of an anchor (although it was Dexter that actually named it anchor, the very first Intermedia papers use the term block) to be an arbitrary selection within any readable document. According to the Intermedia philosophy, creating links should be as easy and as generic as the cut/copy/paste metaphor of desktop environments and thus must be supported by operations in the operating systems underlying those desktop environments. These operations are collected in a so-called linking protocol, a theme that returns in the discussion of open hypermedia systems.

Webs, or the Separation of Structure from Data

Another important concept in the design of Intermedia was that of a web. The basic assumption is that one document (i.e. a node in the generic hypertext terminology) can be part of different navigation networks and that the reader must be able to switch those navigation networks freely, depending on the reader's context. In Intermedia terminology, one such navigation network is called a web and the Intermedia system had special 'Web Views' containing automatically generated layouts of the network structure.

The reason why the web concept is so important, is that it implies that a hypermedia system should "get the links out of the data" or stated otherwise "hypermedia separates structure from data". This principle has proven so valuable that it is almost standard in the design of current hypermedia systems. Nevertheless, the world-wide web ignored this principle with its notion of embedded links, which explains most of the critique of the hypermedia community. We come back to this critique in the section on open hypermedia (p.22).

Object-Oriented Software Engineering

The final justification for discussing Intermedia is that the project advocated for the main theme of this dissertation: the cross-fertilisation between object-oriented software engineering and hypermedia. Intermedia [Meyrowitz'86] was actually a hypermedia specialisation of the fairly general MacApp framework [Schmucker'86] and promoted object-oriented databases as back-end for hypermedia storage [Smith,Zdonik'87].

Both NoteCards and Intermedia are stand alone system that store structures of information. However, they target different application domains and concerning other characteristics they are quite different. NoteCards was there to support individuals in structuring information, so the emphasis was on authoring and browsing tools. Intermedia was designed to support small groups working on shared material so here issues fine-grained concurrency control, tool integration were considered very important.

The research agendas in figure 1 give a glimpse of the problems with this generation of systems and most of them are tackled in the three streams of research that live in the hypermedia field today. But before discussing the issues and contributions of these three streams of research, we mention two miscellaneous hypermedia projects that play an important role in the remainder of this dissertation.

c) Miscellaneous — HyperCard and Dexter

This overview would be incomplete without discussing the HyperCard software [Apple'89] and the Dexter Hypertext Reference Model [Halasz,Schwartz'90]. We classify them in a miscellaneous category as neither of the two is to be considered a real hypermedia system.

HyperCard

HyperCard is to be considered the first really popular hypermedia product in the world and is certainly the first software product that introduced hypermedia to the broader public. There are several reasons for this popularity. First of all, it was bundled for free with every Macintosh computer, so all Macintosh users had access to it. Second, numerous 'stackware' applications were available for almost any problem domain, if not for free then certainly at very low prices. And finally, the product was designed so that anyone who could use a computer could read and create 'stacks'. (Note that more or less the same ideas are part of the world-wide web success story).

The Componentware Approach to Hypermedia

HyperCard is strongly based on a card metaphor. Nodes are called cards and are collected in a stack to form a single hypertext. Nodes take up a fixed size on the screen and can contain several widgets containing text, drawings, …. The behaviour of these widgets is controlled by means of scripts written in the HyperTalk scripting language. There is no explicit notion of links, but some HyperTalk statements can achieve hypermedia navigation effects (i.e. 'goto card …').

HyperCard must be mentioned in this overview because of its end-user tailorability. HyperCard really advanced the state of the art in hypermedia authoring tools, by offering end users an easy-to-use graphical programming environment including a powerful yet easy to learn scripting language. In that sense, HyperCard is actually an ancestor of the componentware approach for building software [Udell'94]: provide a library of powerful customisable components and let end-users assemble their own applications with it. Compared to object-oriented software engineering, the componentware approach is somehow the other side of a spectrum, hence we find it so important.

Is HyperCard a Hypermedia System ?

HyperCard is a good example of the implications of using a definition based on nodes and links like found in [Halasz'87], or a definition that emphasises on the style of building systems like the one we adopted from [Halasz'91].

Despite of its success (or maybe because of its success ?) some hypermedia researchers (see [FountainEtAl'90], [Mylonas,Heath'90]) doubt whether HyperCard is a full fledged hypermedia system[1], among others because it has no explicit notion of links. However, the definition we adopted classifies HyperCard as a hypermedia system, because it offers a degree of end-user customisability rarely met in other hypermedia software.

Nevertheless, because all HyperCard links are computed in scripts and because these scripts reside within the cards, the HyperCard design ignores the principle of "getting the links out of the data". Obeying this principle is not mandatory for being a hypermedia system, but it is definitely a help for providing extra navigational aids (see among others [Lai,Manber'91]).

The Dexter Hypertext Reference Model

The Dexter Hypertext Reference Model [Halasz,Schwartz'90] (Dexter for short) was the result of two small workshops on hypertext with representatives from many of the major existing hypermedia systems. The model attempted to capture the state of the art of that time's most prominent hypermedia systems (i.e. Augment, Document Examiner, IGD, FRESS, Intermedia, HyperCard, Hyperties, KMS/ZOG, Neptune/HAM, NoteCards, Sun Link Service and Textnet), systems belonging mainly to the generation of monolithic hypermedia systems. One of the original goals of the Dexter Hypertext Reference model was to define a hypermedia interchange standard for a wide range of existing and future hypermedia systems, and although this goal was never realised, the Dexter model had a considerable impact on subsequent hypermedia research; some hypermedia systems even initiated their design using the Dexter specification (notably DeVise hypermedia-DHM [Grønbaek,Trigg'94] and the Amsterdam Hypermedia Model-AHM [Hardman,Bulterman,VanRossum'94]).

The design of the Zypher framework was based on the Dexter Model as well, and we refer to Chapter 3 (Data Structures for an Interoperable Hypermedia Framework- p.76) for a more detailed discussion of the Dexter Model. Here we focus on the important ideas.

Separation of Concerns: Storage Layer, Within-Component Layer & Run-time Layer

First, by adapting the generic hypertext system architecture of the Hypertext Abstract Machine (HAM) model [Campbell,Goodman'87] Dexter emphasised the separation of concerns important in hypermedia system design. As depicted in figure 2, Dexter proposes three layers: the within-component layer (mostly referred to as the within layer), the storage layer and the run-time layer. We quote the original paper [Halasz,Schwartz90] to define the responsibilities of each layer.

---

1 According to [Nielsen'90], p. 93, Bill Atkinson (the designer of HyperCard) has admitted that it was not really designed as a hypertext product.

The storage layer "*models the basic node/link network structure that is the essence of hypertext. […] The storage layer focuses on the mechanisms by which link and non link components are 'glued together' to form hypertext networks. The components in this layer are treated as generic containers of data. No attempt is made to model any structure within the container*". The within-component layer "*is specifically concerned with the contents and structure within the components of the hypertext network*". The run-time layer provides "*tools for the user to access, view and manipulate the network structure*".

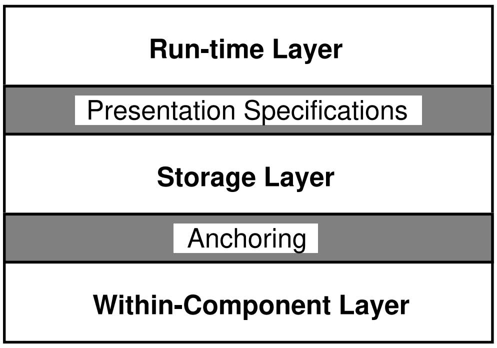


Figure 2: Dexter Model Layers

The emphasis of the Dexter model is on the storage layer and the two adjacent protocols. Using these two protocols, Dexter specifies a standard way of constructing hypermedia structures and storing them into an underlying "hyperbase". The within-component layer would rely on existing standards such as SGML and HyTime, while the run-time layer is the responsibility of any (hypermedia) application that adheres to the Dexter protocol.

Careful analysis of the definitions of the layers reveals that the separation of concerns between the storage and within-component layers is actually a rephrasement of the "get the links out of the data" principle. Also, the separation of concerns between the storage and run-time layer is actually a preparation for the open hypermedia philosophy, where any desktop application is allowed to manipulate the hypermedia navigation network.

The Anchor concept

Stating that the storage layer should be separated from the within-component layer is one thing, but then how should both layers interface with each other ? A hypertext system obeying the principle of "separating structure from data" requires a mechanism for addressing locations within that data. It was a major contribution of the Dexter model to formalise the concept of anchors (borrowed from the Intermedia project) for that purpose. Dexter also proposed the concept of presentation specifications as the glue between the storage layer and the presentation layer, but the hypermedia community did not pick up the idea.

We classified HyperCard an Dexter in a miscellaneous category, but again they target a different application domain and so we can expect some fundamental differences between the two as well. HyperCard aimed to support an individual in organising and presenting information, so the emphasis was on authoring and presentation. Dexter was mainly influenced by hypermedia systems supporting small groups of people working on shared material. Hence the emphasis on the storage layer, which among others enables to incorporate concurrency and version control.

d) The Open Hypermedia Systems — DHM and Microcosm

A first observation about the generation of monolithic hypermedia systems is that they are not 'open' in the sense that only applications part of the hypermedia system are able to process the information inside the navigation network. As a result, hypermedia systems are used for specialised tasks only; not in the daily information processing since people use other tools there (e-mail, text-processors, …) [Meyrowitz'89], [Brown'90].

The above observation gave rise to a stream of research generally referred to as open hypermedia. One of the best specifications of what open hypermedia research encompasses has been formulated in a scenario where engineers from the Boeing Airplane Manufacturing Company propose a hypermedia system as a way of integrating data, tools and services that support engineer's work practices [Malcolm,Poltrock,Shuler'91]. We come back to this scenario when we discuss open hypermedia (p.22). Here we discuss two representative examples of well-known open hypermedia Systems, and name most of the other known hypermedia systems.

Microcosm: an example of a link service raising the interoperability issue

Microcosm [FountainEtAl'90] is one of the first hypermedia systems to incorporate interoperability standards provided in the operating system (their first version used DDE - Dynamic Document Exchange under Windows) to allow external applications to make use of the linking facilities provided by the hypermedia system. Such an approach is typically called a link service (see also [Pearl'89]). Microcosm comes along with several built-in document viewers and the project investigated several ways to make external applications aware of the presence of the link service ([Hall,Hill,Davis'93], [DavisEtAl'92], [Davis,Knight,Hall'94]).

An interesting remark about the Microcosm project is that they have built up a notable reputation in dealing with multi-media archives. Among others by building electronic versions of the Mountbatten and Churchill archives. Noteworthy is also that, after a long period of experimentation in University labs, Microcosm has recently become a commercial product.

Generic Links and Filters

Generic links are an important ingredient of the Microcosm philosophy. The Microcosm project values links as an important store of knowledge besides the documents themselves. If the links are bound too closely to the documents, they cannot be applied to new documents. The similarities with the web concept in the Intermedia project are striking and indeed Microcosm is a strong advocate for the "get the links out of the data" principle [Davis'95].

Microcosm takes a slightly different approach however. Besides specific links between particular locations inside specific documents, the Microcosm system has this notion of a generic link, which once authored from a certain pattern (be it a text string or a piece of multi-media information [LewisEtAl.'96]) to a destination anchor, may be followed from any occurrence of the pattern in any document. Following generic links makes the system search the link base for all matches of a certain pattern to compute a list of possible target locations. Users can control this search by plugging filter modules, where each filter represents a special matching/searching algorithm.

The Componentware Approach to Open Hypermedia

Microcosm can be regarded as representing the componentware approach to open hypermedia. In essence, the Microcosm system is a link service for a collection of tailorable modules to be assembled in a hypermedia environment. There exist two types of modules: viewer applications and filters.

Microcosm classifies the viewers in three categories. Fully aware viewer applications [Hall,Hill,Davis'93] are viewers where the source code of the viewer was available and has been enhanced to include calls to the Microcosm API. The more interesting categories are the semi-aware and universal viewers [Davis,Knight,Hall'94]. Semi-aware viewers are third party applications that can be tailored (by means of a scripting language or macros) to include at least part of the Microcosm functionality. The generic example for these kind of viewers is Microsoft Word, which can be extended quite easily using the VisualBasic scripting language and the Microsoft Windows DDE inter-application communication facilities. The universal viewer is the means to handle applications that cannot otherwise be integrated with the Microcosm link service. It is a kind of a parasitical program (a "shim" in Microcosm terminology) that is wrapped around an unaware application and handles communication between the Microcosm link service and the application.

Filters are the modules that situate themselves on the other side of the link service. When a viewer application passes a message to the Microcosm link service, the message is passed through a chain of appropriate filters. Each filter decides what actions to take and returns messages to the link service which in turn passes them to the appropriate viewer applications.

Besides the tailoring facilities offered by viewer applications and filters, there is yet another tailorability dimension and that is the communication protocol between the different modules. There, Microcosm adheres to a free-format message passing protocol where modules communicate by exchanging messages containing lists of slots. The list of possible slots is unrestricted and receivers of the messages are not forced to interpret all the slots in order to understand the message. This idea is now taken as the basis for a definition of a standard communication protocol between open hypermedia systems. Below is shown how the current proposal [Davis,Lewis,Rizk'96] would request for opening a document called "OHP.htm" in read-only mode. The "DocumentNickName" slot is an example of an optional slot that can be used to provide redundant descriptions of the document.

```
\Subject LaunchDocument
\DocumentName "OHP.htm"
\ReadOnly True
\DocumentNickName "The OHP Protocol"
\DocumentType HTML
\DataCallBack False
\Channel 40
```

DHM (DeVise Hypermedia): The Framework Approach to Open Hypermedia

DHM [Grønbaek,Trigg'94] is an open hypermedia system designed to support co-operative design in (among others) large engineering projects. In contrast with Microcosm, DHM is more oriented towards object-oriented software engineering techniques to provide the kind of openness demanded in engineering environments.

As such, the DHM project incorporates object-oriented database and framework technology (again, we see the influence of the Intermedia project). Also, DHM allows end-users to extend the system by providing a kind of access to the source code level without the need to employ the full development environment [Grønbaek,Malhotra'94]. This is actually an improvement over the API approach taken by NoteCards or the framework approach taken by Intermedia.

An important distinction between DHM and Microcosm is that Microcosm enforces viewer applications to store their own documents, while DHM provides an extra service to the viewer applications by allowing them to store the documents inside the hypermedia system. The latter is very important in collaborative settings to handle integrity, version control and change notification.

DHM and Dexter

Besides being an open hypermedia system, DHM has an outstanding reputation concerning extensions to the Dexter Reference Model [Grønbaek,Trigg'94], [Grønbaek'94], [Grønbaek,Trigg'96].

Dexter was an attempt to capture the design of the generation of monolithic hypermedia systems, but was underspecified to play its role as a reference model for exchange of hypermedia. DHM has proposed some extensions and modifications, among others concerning dangling links, link directionality, anchoring within composites [Grønbaek,Trigg'94]; locking and notification [GrønbaekEtAl.'94]; composites [Grønbaek'94]; location specifiers and reference specifiers to deal with embedded references, anchors in external documents and dynamic hypermedia structures

[Grønbaek,Trigg'96]. The DHM project has proven that the Dexter model —although a reference model for monolithic hypermedia systems— is a good basis to build open hypermedia systems.

Other Open Hypermedia Systems

Since open hypermedia is one of the main themes of this dissertation, we mention the other systems and projects that should be reckoned as open hypermedia. Sun's Link Service [Pearl'89] is a first attempt to incorporate a linking protocol in the operating system so that any application can make use of it. MultiCard is an open hypermedia system somewhere in between Microcosm and DHM. With Microcosm it emphasises on an extensible linking protocol that enables third party applications to access the link service; like DHM it offers hyperbase functionality with concurrency control features. HyperTED [Vanzyl'94] is an open hypermedia system that experiments with features specific to particular application domains to see whether this may help in solving open hypermedia problems. Hyperform [Wiil,Legget'92], [Wiil,Legget'93] and its successor Hyperdisco [Wiil,Legget'96] is much like DHM in the sense that is also based on object-oriented techniques to extend the system. One of the appealing features of Hyperform is the way it allows to experiment with various approaches for concurrency control and notification. The Hypermedia Research Laboratory in Texas developed a series of hyperbase and hypersystem prototypes called HB_{0-3}/SP_{0-3} [Kacmar,Legget'91], [Legget,Schnase'94] with the latest descendant named HOSS [NürnbergEtAl'96]. These prototypes stress the importance of processes in an open hypermedia system and lately started to explore potential benefits of having maximal operating system support for hypermedia functionality. Chimera [Anderson,Taylor,Whitehead'94] is —to our knowledge— the only system that demonstrates how third party applications can be used to enrich software development environments.

Microcosm and DHM are both open hypermedia systems. The reason they have chosen a different approach is because they aim to support a different application domains. Microcosm wants to support individuals working with various kinds of information, hence the emphasis on tools for authoring links and on customisability. DHM supports co-operative design in large engineering projects, so the emphasis is more on concurrency control, sharing of information and uniform tool enhancements.

e) The Distributed Hypermedia Systems — ABC and WWW

A second observation about the generation of monolithic hypermedia systems is the lack of 'distribution', in the sense that all the information inside the navigation network is stored in one centralised repository. In most cases, the centralised repository is a file system, although some hyperbases apply database technology as well (i.e. [Schütt,Streitz'90], [Wiil,Legget'92], [Wiil,Legget'93], [Wiil,Legget'96]). To some extent, the upcoming distributed database technology may serve the needs of distributed hypermedia. However, to implement the Xanadu vision, one must leave the notion of a centralised repository altogether, as some hypermedia systems do. The latter are kind are classified as the distributed hypermedia systems.

WWW: World-Wide Web

The hypermedia project coming nearest to implementing the Xanadu vision was the world-wide web project [Berners-LeeEtAl'94]. The world-wide web consists of an extensible set of "page-servers", where each page contains a number of embedded references to other pages residing on (often remote) servers. Web surfers (jargon for users reading world-wide web pages) employ browsers (the client programs) to navigate the loosely coupled graph of web pages by loading these pages and activating the embedded references to load referenced documents. The world-wide web utilises the Internet wide area network to connect all those servers, so that users really experience having the world at their fingertips: with a single mouse click they can jump to the other side of the world.

The world-wide web was developed at CERN as a medium to exchange scientific results, but rapidly reached momentum outside the scientific community and experienced an exponential growth of running servers and clients. Most software vendors, certainly the very large ones like Microsoft and ORACLE, adapt their strategies and software to incorporate some kind of web functionality.

The HTML, HTTP, URL Acronyms

There are a number of acronyms often used to refer to the world-wide web, but are in fact but technological cornerstones for enabling the establishment of such a large scale distributed hypermedia environment.

HTML (Hypertext Mark-up Language) is an interchange standard used to submit pages from the server to the clients. The idea is that an ordinary text document includes several "mark-ups" that tell the client how to format it. HTML includes mark-up for titles, boldfacing, graphics and —most important— anchors. Anchor mark-up contains the embedded references to the other pages and is supposed to be highlighted by client programs (typically in some colour and underline). HTML is defined as a growing standard, in the sense that future versions of the HTML standard are supposed to embody the older versions.

HTTP (Hypertext Transfer Protocol) is the network protocol most commonly used to transfer pages. These pages may be in HTML format, but HTTP can handle other formats as well (the HTTP protocol contains limited format negotiation). An important limitation (although essential on such scale) of the HTTP protocol is that it is stateless, i.e. the network connection is only held for the time needed to transfer the page. This makes it especially hard to use the world-wide web in collaborative settings. Note that most web browsers adopt other network protocols like FTP (file transfer), SMTP (electronic mail) as well.

URL (Universal Resource Locator) is the name for the format of a page address, which is used in the embedded references and anchor mark-up. The URL is actually an extensible addressing format, which makes it easy to refer to new types of page servers. A URL is prefixed with a key word identifying the address space of the referenced page (i.e. 'file' for file servers, 'http' for HTTP servers, 'ftp' for FTP servers, 'mailto' for electronic mail, …). The format of the rest of the address depends entirely on this prefix, but consists in most occasions of a set of identifiers separated by "/" (a hierarchical level) or "?" (an argument to a query).

Most world-wide web servers map HTTP requests to HTML documents residing on some file system. However, the HTTP protocol and the HTML format are sufficiently simple to be easily mimicked by other programs. For example, it is quite easy to set-up a program that translates user requests into database queries and returns the result in HTML format. Or a server may decide to store documents in another format and generate HTML on the fly with each request.

The diversity of protocols and formats and the ability to generate everything on the fly makes it hard to discuss the web as a hypermedia system. For example, the Web is frequently criticised as not respecting the "separate data from structure principle" (for using embedded references), while this is counter argued by saying that a server may very well store the links apart from the data and embed the links only when transferring the page (i.e. like in the Hyper-G [Andrews,Kappe,Maurer'95] and HOME [DuvalEtAl95] projects).

ABC: Artefact Based Collaboration

ABC is a project that has taken into account the issue of distribution from the very beginning. To deal with large scale hypermedia (such as in the aerospace industry

[Malcolm,Poltrock,Shuler'91]), "the architecture must permit distribution across multiple hardware platforms; to be distributable, the data model must be partitionable into objects that tend to be accessed separately and that have relatively few links and/or dependencies with other objects." [Smith,Smith'91]. Besides nodes and links, the ABC design is based on the notion of subgraphs, providing a natural way to partition the overall graph structure and distribute it over a number of graph servers. The ABC system comes with a multitude of network browsers that provide end-users with the necessary tools to manipulate the complex navigation structures.

The striking difference with the world-wide web project is the support for collaboration. The ABC project has investigated many issues related with sharing material over distributed servers, such as access control and concurrent access [Schackelford,Smith,Smith'93].

ABC is also an open hypermedia in the sense that the primitive nodes in the graph structure contain files to be processed by third party applications. ABC takes the viewpoint that hypermedia graph servers should become the file systems of the next generation of operating systems [Smith'96] and that hypermedia navigation will be an essential ingredient of the next generation operating systems (i.e. the graph browsers will replace the file managers of today's desktop environments). With the growing interest for world-wide web technology (among others by operating system vendors like Microsoft) this seems to be very promising.

Both being representatives of distributed hypermedia systems, the world-wide web and ABC are quite different with respect to the target application domain. The world-wide web aims to support individuals in browsing and publishing material and most work is on the development of interchange standards and communication protocols. ABC is about supporting groups, so incorporates various concurrency control mechanisms.

f) The Spatial Hypermedia Systems — Aquanet and HyperCafe

A third observation about the generation of monolithic hypermedia systems is the poor performance when dealing with 'dynamic' structures. Most hypermedia systems perform well for information presentation, but force authors to specify all navigation relationships explicitly by hand. So the support for structuring information can be improved, certainly when it comes to dynamic structures.

Aquanet and its successor **VIKI**

Although there were other systems that abandoned the basic node/link model to deal adequately with dynamic structures (i.e. String searching in SuperBook [Remde,Gomez,Landauer'87]; Petri nets in Trellis [Furuta,Stotts'89]; Taxonomies [Parunak'91]) it was with the Aquanet project [MarshallEtAl'91] that the Spatial Hypertext research took off.

The idea underlying the Aquanet project is to support a group in structuring collective knowledge by providing multiple views on the same knowledge structure. However, unlike traditional hypertext approaches, users are not forced to make these knowledge structures explicit by connecting nodes with typed (labelled) links (i.e. gIBIS [Conklin,Begeman'88]). In Aquanet, relationships are inferred from the spatial organisation of nodes [Marshall,Shipman'93]. VIKI [Marshall,Shipman,Coombs'94] and [Marshall,Shipman'95], which is the successor of Aquanet, allows to formalise those emerging structures on user's demand.

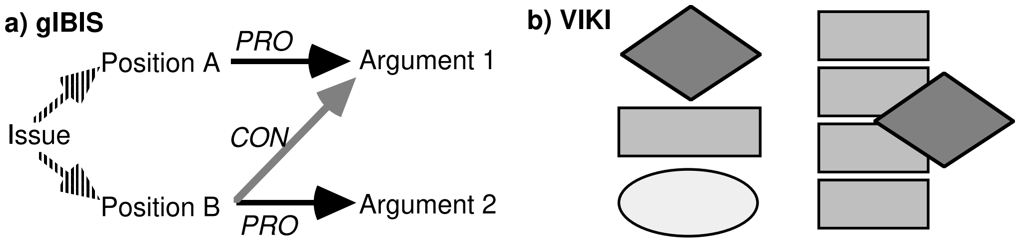


Figure 3: Traditional Hypertext (a) versus Spatial Hypertext (b)

Figure 3 illustrates the difference between a traditional way of creating a hypertext structure and the spatial way. On the left we see part of a gIBIS network stating that there are two possible positions for a given issue and that there are two arguments sustaining the positions; moreover one argument not only sustains one position but also counters the other position. The create such an argumentation structure, a user creates five nodes (one issue, two positions and two arguments) and five links. Note that creating a link is a complex operation, one must choose the type of the link, mark its source and target an provide an optional label.

To the right of figure 3, we see two VIKI structures: one with a diamond, rectangle and oval denotes a composite relation; the other with four rectangles and a diamond denotes a labelled relation. This means that to create a composite relation, a user must place three node with three different types one beneath the other. And if a user places a group of nodes with same type one beneath the other, and an extra node with different type on top of it this is recognised as a labelled relation. So, creating relationships with VIKI requires less steps than traditional hypertext approaches.

HyperCafe

Aquanet and VIKI emphasise the idea of structuring knowledge and somehow lose the idea of navigation and multi-media. The HyperCafe project [Sawhney,Balcom,Smith'96][2] reincorporates both notions into the spatial hypertext research. HyperCafe is designed as a cinematic experience of hyper-linked video scenes; the metaphor is that of a human visiting a cafe and picking parts of the stories told by all the other visitors. While the users follow a story in the foreground of the video, the background shows characters telling other stories to each other. Users can select those background characters to switch stories. So far, this is navigation in video, but the trick is that users can choose to run multiple foreground stories by arranging several simultaneously running videos on the screen. Such spatial organisation produces a whole new story, hence the classification as spatial hypermedia.

Why is this experiment worth mentioning ? Remember that, according to our definition, hypermedia is a "style" rather than a number of features and that style is about freedom - about customisability. However, there is not so much freedom for "readers" of traditional hypermedia, as they can only follow the links provided by the authors. HyperCafe shows an extra degree of freedom, where users can create stories that were not intended by the authors.

Again we see that the differences between the two systems stem from the target application domain. The Aquanet and VIKI design are based on the "WYSIWID" principle, which is an acronym for "What You See Is What I Did". The systems supports co-operation and the reason they want easy authoring is because they want to support discussion in little groups. HyperCafe maximises the customisability of individuals and there the emphasis is on the presentation of information.

2 This paper was rewarded with the first Douglas Engelbart award at the Hypertext'96 Conference [HT'96].

Conclusion

The purpose of this section was to confront the reader with the important ideas that live (and lived) in the hypertext and hypermedia community. The section included a definition of hypermedia as a combination of navigation and a style. Navigation is about browsing non linear information structures. Style emphasises on a feeling of freedom, in the sense that users must be able to customise the presentation of information in some way. The definition comes with some basic vocabulary (i.e. node, link and anchor).

The important ideas were illustrated with an overview of existing hypermedia systems, roughly classified as pioneering, monolithic, miscellaneous, open, distributed and spatial systems. The overview emphasised two threads important to the remainder of the dissertation: tailorability and the correct separation of concerns. The latter stresses the "get the links out of the data" principle and was illustrated with the Dexter hypertext reference model. The former was motivated by the observation that each hypermedia is designed for another application domain and was presented as a spectrum ranging from support for individuals as opposed to groups and componentware as opposed to object-oriented software engineering.

Open Hypermedia

While the previous section provides us with a general introduction of hypermedia, this section discusses the particular subfield called open hypermedia systems. Open hypermedia research starts from the assumption that to provide real support for daily information processing, hypermedia systems must integrate external applications to manipulate information. Several compelling issues pop up when dealing with this integration requirement.

Open hypermedia research is quite an active subfield. Since 1989, the hypermedia conferences accepted several papers discussing approaches for open hypermedia and already three workshops were organised to exchange ideas (i.e. the Konstanz workshop [Aßfalg'94]; the 1st Workshop on Open Hypermedia Systems [Wiil,Østerbye'94]; the 2nd Workshop on Open Hypermedia Systems [Wiil,Demeyer'96]).

As always within an active field of research, it is quite hard to give a precise definition. Here we present different approaches, which should give a good overview of what researchers mean when speaking of open hypermedia systems. Several authors have attempted to define an open hypermedia system, but most definitions fall too short and are supplemented with requirement lists. As such requirement lists tend to be vague and imprecise, there is a growing tendency to write scenarios describing how these requirements fit into the usage of an open hypermedia system for the daily work of a group of people performing a particular task. Finally, we present the Flag taxonomy as a first attempt to classify open hypermedia approaches rather than systems.

Open Hypermedia Definition & Requirements

We adopt the most recent definition of an open hypermedia system, as we agree that it gives a reasonable summary of current thinking.

"*An open hypermedia system is a hypermedia system, where the term open implies the possibility of importing new objects into a system. A truly open hypermedia system should be open with regard to:*

1. *Size: It should be possible to import new nodes, links, anchors and other hypermedia objects without any limitation to the size of the objects or to the maximum number of such objects that the system may contain, being imposed by the hypermedia system.*
2. *Data Formats: The system should allow the import and use of any data format, including temporal media.*
3. *Applications: The system should allow any application to access the link service in order to participate in the hypermedia functionality.*
4. *Data Models: The hypermedia system should not impose a single view of what constitutes a hypermedia data model, but should be configurable and extensible so that new hypermedia data models may be incorporated. It should thus be possible to interoperate with external hypermedia systems, and to exchange data with external systems.*
5. *Platforms: It should be possible to implement the system on multiple distributed platforms.*

6. *Users: The system must support multiple users, and allow each user to maintain their own private view of the objects in the system.*" [Davis,Lewis,Rizk'96].

We elaborate on the requirement list part of the definition by combining the following sources[3]: the 7-issues research agendas [Halasz'87] and [Halasz'91]; the Boeing paper [Malcolm,Poltrock,Shuler'91], reiterated in [Poltrock'96]; a paper from the HB/SP series of systems [Legget,Schnase'94]; the preamble to the Konstanz Open Hypermedia Workshop Proceedings [Aßfalg'94]; papers from the Microcosm group [DavisEtAl'92] and [Hill,Hall'94]; a paper from the HyperTED project [Vanzyl'94]; a paper from the Hyperdisco project [Wiil,Legget'96] and finally —the one that is part of our open hypermedia definition and serves as the backbone— the paper on the Open Hypermedia Protocol [Davis,Lewis,Rizk'96].

OHS-REQ 1) Size

It should be possible to import new nodes, links, anchors and other hypermedia objects without any limitation to the size of the objects or to the maximum number of such objects that the system may contain, being imposed by the hypermedia system.

The size issue covers two different aspects: the size of an individual object and the number of different objects in the system. The former is a typical problem when dealing with multi-media data. The latter is crucial when dealing with large scale hypermedia systems (i.e. Boeing speaks in the order of 1 million nodes and 10 million links) or distributed systems (i.e. the world-wide web).

This issue of size has some important implications. First of all, size implies the possibility to access data residing outside the hypermedia system, since large object sizes may prevent copying the data into the system. Secondly, size affects scalability and thus addressability. A system having unlimited capacity needs at least an unlimited and scalable address space. Furthermore, as one cannot expect to know the sources of external data in advance, the address space must be extensible.

Storing information outside the hypermedia system challenges the design of the hypermedia system with respect to consistency. For example, one cannot ensure link consistency since data can be edited without informing the hypermedia system (this is called the editing problem). Link robustness demands for techniques and heuristics to fix corrupted link structures. Also, data residing outside the hypermedia system implies that the original data must remain unchanged (this is called document sanctity), since one cannot be sure to have write-access to the information (i.e. the data can reside on read-only media such as a CD-ROM). Document sanctity is another argument for the "get the links out of the data" principle [Davis'95].

OHS-REQ 2) Data Formats

The system should allow the import and use of any data format, including temporal media.

This can be read as a multi-media requirement, although it covers issues like legacy data formats and encryption techniques as well. The requirement implies at least the ability to launch another application that knows how to interpret the data format, since it is impossible to build a single hypermedia system that knows how to interpret all current and future data formats. Standards (see OHS-REQ 4) make life easier of course.

An implication not immediately visible from this requirement (but follows from combining it with part of OHS-REQ 4, exchange data with external systems) is that a

3 The original idea to combine open hypermedia requirement lists came from Helen Ashman in a draft paper investigating how well the World-Wide Web performs on each of these requirements. As may be surprising at first, for many of the requirements the Web performs better than the proclaimed open hypermedia systems.

hypermedia system should be able to deal with virtual data as well. Virtual data is not available from some persistent storage but is generated by some external device or program; typical examples are video conferencing, database queries and simulation tools. Note that both temporal and virtual data impose special problems on hypermedia systems, like how can one specify anchors inside temporal or virtual data.

OHS-REQ 3) Applications

The system should allow any application to access the link service in order to participate in the hypermedia functionality.

No doubt this is a mandatory requirement for an open hypermedia system, raising the interoperability issue (i.e. making applications work together without explicit knowledge of each other). Lots of techniques are available to extend "any" application to incorporate hypermedia functionality, among others described in the HB_0/SP_0 (or Proxhy) paper ([Kacmar,Legget'91]), the Microcosm papers ([DavisEtAl'92], [Hall,Hill,Davis'93], [Davis,Knight,Hall'94]) and the HyperTED paper ([Vanzyl'94]).

This requirement states that an open hypermedia system should have a link protocol, accessible by any other application that wants to use it. This is the opposite of having the hypermedia system launch another application with a given set of data (as implied by OHS-REQ 2).

OHS-REQ 4) Data Models

The hypermedia system should not impose a single view of what constitutes a hypermedia data model, but should be configurable and extensible so that new hypermedia data models may be incorporated. It should thus be possible to interoperate with external hypermedia systems, and to exchange data with external systems.

Given the enormous variety of data models existing in the hypermedia field, this requirement is very hard to realise (conceive an open hypermedia system that is able to deal with the basic node-link model as well as a data model supporting spatial hypertexts). What makes it especially difficult is that lots of the information is formed by the "link" structure, so that an open hypermedia system must be able to incorporate different link models. Hence the need for an extensible link engine, which is free to choose the linking model that best suits the needs of the information structure it represents. Microcosm (with its generic links and filters) and DHM (with its location and reference specifiers) provide building blocks for such an extensible link engine.

A first related issue is that of standards: with a standard hypermedia reference model one could exchange data with other hypermedia systems without having to write translators for all possible combinations. Note that [Legget,Schnase'94] reports that a possible disadvantage of standards is the loss of structure. As far as data formats concern, hypermedia can rely on existing standardisation efforts such as HyTime [Newcomb,Kipp,Newcomb'91]. For the communication aspects, the open hypermedia community is developing a standard communication protocol for link services [Davis,Lewis,Rizk'96].

The second issue is that of extensibility, which brings us back to the style aspect in our hypertext and hypermedia definition (p.08). As hypermedia style emphasises a feeling of freedom and a degree of customisability, extensibility plays a crucial role. This is not a problem of the hypermedia field alone and there are lots of techniques available that can be applied in various combinations.

- Configuration tools: Special purpose extension tools, most of them with user-friendly graphical interfaces. Examples are the "Preferences" tools coming with most of today's desktop applications and the control panels in the Macintosh and Windows operating systems. In the hypermedia community, HyperCard is still the most prominent example of how configurable hypermedia software can be.

- Scripting languages & macro facilities: Programming languages aiming at skilled end users. Examples are HyperTalk (HyperCard's scripting language) and VisualBasic (the scripting language coming with the Microsoft Office suite of products).
- Dedicated programming languages: Full fledged programming language dedicated to extend one type of systems. Examples: Shell languages for operating systems; SQL; Java [Java'95]; MC2000 [Rizk,Sauter'92].
- General purpose programming languages: A full fledged programming language used for other purposes than extending a particular system. Example: Lisp in AutoCAD and GNU Emacs; Scheme in Hyperform [Wiil,Legget'92].
- Object-oriented frameworks: A reusable design for a specific problem domain implemented in some object-oriented medium and extended by plugging new objects. Example: Intermedia [Meyrowitz'86] and DHM [Grønbaek,Trigg'94].
- Open Implementation: A computational system with the ability to manipulate its own computations. Examples: Silica [Rao'91]; ApplFLab [SteyaertEtAl'94] & [SteyaertEtAl'95].
- Extensible data models: Systems with a data model that can easily be extended for certain purposes. Examples: filters in Microcosm [DavisEtAl'92]; hyperdocuments in [DeBra,Houben,Kornatzky'92].

OHS-REQ 5) Platforms

It should be possible to implement the system on multiple distributed platforms.

Depending on the scale of distribution, systems must incorporate a range of techniques to ensure fast response times (i.e. data caching, prefetching). To produce adequate results the system must gather knowledge about often followed navigation trails, usage patterns, ...

Also, distributed hypermedia systems raise the document sanctity issue and the editing problem (see OHS-REQ 1). Yet, the scope of possible solutions is completely different as consistency seems to be less crucial in large scale distributed systems such as the world-wide web. One may adopt some local consistency scheme complemented with periodic "garbage collection".

Supporting multiple platforms raises the issue of portability across different soft- and hardware platforms, an issue that overlaps with the data formats (OHS-REQ 2), applications (OHS-REQ 3) and data models (OHS-REQ 4) requirements. Indeed, there is a multitude of data formats, applications and data models and the choosing the best depends upon the intended soft- and hardware platform. One way to deal with portability across different soft- and hardware platforms is to incorporate some form of protocol negotiation, where part of the communication between modules is about choosing the optimal configuration.

OHS-REQ 6) Users

The system must support multiple users, and allow each user to maintain their own private view of the objects in the system.

Hypermedia has always played the role of enabling technology for computer supported co-operative work. This requirement handles two aspects related to this role: one of information sharing and one of information perspectives.

If multiple users work with the same hypermedia structure, then the hypermedia system must control access to the shared data. Authority control is the process that verifies whether a user is allowed to read or modify data. Generally, an open hypermedia system should not have an artificial distinction between authors (allowed to modify) and readers (read only access), such as imposed by systems having a separate authoring environment. Concurrency control deals with the problems of simultaneous access to the same data element and is essentially based on locking (restrict access temporarily to ensure safe data transfer) and notification (warn the other users working with a data element that something has happened).

An important aspect of co-operative work is the ability to create private work spaces. In a hypermedia context this means that (a subgroup of) users must be able to build a personal substructure inside the global hypermedia structure. This is again a plea for the "get the links out of the data" principle, since a shared node may participate in different link structures. This aspect interferes heavily with the linking models supported in the hypermedia structure.

Closely related is the issue of version control (the ability to have several variants of the same data element and/or structure). Many hypermedia researchers agree that versioning is important, yet few systems provide version control and few users seem to complain[4].

The above requirement list makes it very clear that incorporating hypermedia functionality into the everyday working environment involves more than providing link services to third party applications. The closer examination of the items in the list reveals that open hypermedia systems cover themes like interoperability (1-3), extensibility (4) and distribution (5-6), which are central themes in the design of all kinds of open systems — operating systems (i.e. UNIX and OS/2), databases (Exodus and CORBA), inter application communication (OLE, OpenDoc), tailorable software (i.e. Emacs and AutoCAD), programming languages (Smalltalk and CLOS). By adopting this requirement list as the definition for an open hypermedia system, we see that open hypermedia systems are quite representative for the domain of open systems. *Which makes open hypermedia systems an excellent choice for experimenting with techniques to build open systems*, a central corner stone in our argumentation.

Open Hypermedia Scenarios

From its conception (among others in [Yankelovich,Meyrowitz,VanDam'85]), open hypermedia research was aiming to get hypermedia functionality out of the hypermedia systems and into the every day environments people use for working with information. With the "Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise" [Malcolm,Poltrock,Shuler'91] paper, the hypermedia community finally distilled a requirement list out of the working practices of engineers participating in the design and manufacturing process of airplanes. The paper is a milestone in hypermedia research because (quoting Frank Halasz [Halasz'91]) "*a user-centered technology like hypermedia needs users to drive the research*". As the authors of the paper were all associated with Boeing —world's largest producer of commercial airplanes and one of the biggest employers in the US— this could count as a user community.

Another reason why the paper is so appealing, is because it contains a scenario illustrating how the requirement list fits into the everyday working practice of an engineering team. The scenario described how a new team member could learn about a project (navigating through the project notebook), exchange information with other team members (e-mail and tele conferencing), and contribute to the project's goals (preparing reliability analysis) all within an environment seamlessly integrating all the different tools applied by engineers in their day to day work. The open hypermedia community has picked up this scenario idea: it reappeared in [GrønbaekEtAl.'94], [Legget,Schnase'94] and [Davis,Knight,Hall'94] and to some degree in (the presentation accompanying) [Vanzyl'94]. Moreover, the 2nd Workshop on Open Hypermedia Systems [Wiil,Demeyer'96] decided to set-up a web-site collecting and exchanging scenarios for open hypermedia usage (see http://www.csdl.tamu.edu/ohs/).

---

4 Version control is not part of the Boeing requirement lists [Malcolm,Poltrock,Shuler'91] and [Poltrock'96].

The Flag Taxonomy of Open Hypermedia Systems

Although the open hypermedia definition gives a good summary of current thinking, it is too requirement oriented and thus too long. Moreover, it is too strong, since none of the existing open hypermedia systems meet all of the listed requirements. The Flag taxonomy [Østerbye,Wiil'96][5] takes a completely different angle by making an attempt to capture the essence of an open hypermedia system and use them to classify open hypermedia systems.

The Flag taxonomy started of as a "written before the facts" conclusion for the first open hypermedia workshop [Wiil,Østerbye'94]. The workshop organisers prepared a set of slides with a number of diagrams showing many of the design trade-offs involved in the construction of an open hypermedia system. These diagrams were discussed during the workshop and although the preformulated conclusion proved incorrect, it served as an adequate framework for discussing open hypermedia systems. As, the workshop proclaimed the need for an open hypermedia reference model —i.e. much like the Dexter model was for the generation of monolithic hypermedia systems— these diagrams were seen as a good starting point. The flag taxonomy is actually a mixture of those concluding slides with some of the ideas of the Dexter model.

If one looks at the layered architecture of the Dexter model (see left hand side of figure 4) from an open hypermedia perspective, an important difficulty is revealed: the within-component layer is separated from the run-time layer by means of a storage layer. This is a typical symptom for monolithic systems, where a hypermedia system somehow constrains the possible contents of nodes to be presented in the run-time layer, because these node-contents must be stored inside the hypermedia system. However, this disregards a fundamental principle of open hypermedia systems where viewer applications need to store the node contents outside the hypermedia system. So, the Dexter architecture actually violates requirements 1-3 of the open hypermedia definition. To cope with this difficulty one can adapt the Dexter model and pull together the within-component layer and run-time layer. Doing so results in the "Dexter-pie" depicted in the right hand side of figure 4.

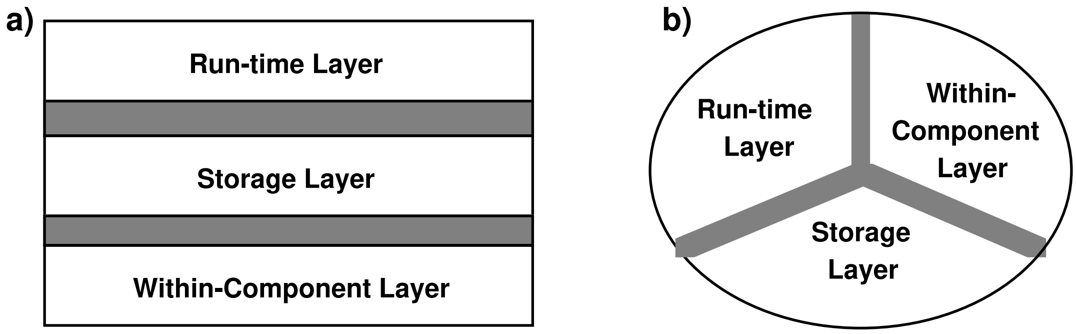


Figure 4: Layered Dexter Architecture (a) versus the Dexter Pie (b)

Remember that there is another important principle in the design of (open) hypermedia systems, which is the "get the links out of the data". In the Dexter model, this principle is captured in the protocols between the layers and this is inherited by the Dexter pie. The flag[6] taxonomy goes one step further, as it captures those two essential separations of concerns for an open hypermedia system by performing one additional cut (see figure 5). The horizontal cut separates contents from structure (i.e. data from links) while the vertical cut separates storage from run-time. This results in four functional modules, which the authors named storage manager, viewer, data model manager and session manager. Also there are four protocols between functional modules. Note that the protocol of prime interest in open

---

5 This paper was one of the nominees for the first Douglas Engelbart award at the Hypertext'96 Conference [HT'96].

6 A glance at Figure 4 combined with the Danish nationality of the authors reveals the inspiration for the name.

hypermedia research is the linking protocol between the viewer and the session manager (see [Davis,Lewis,Rizk'96]).

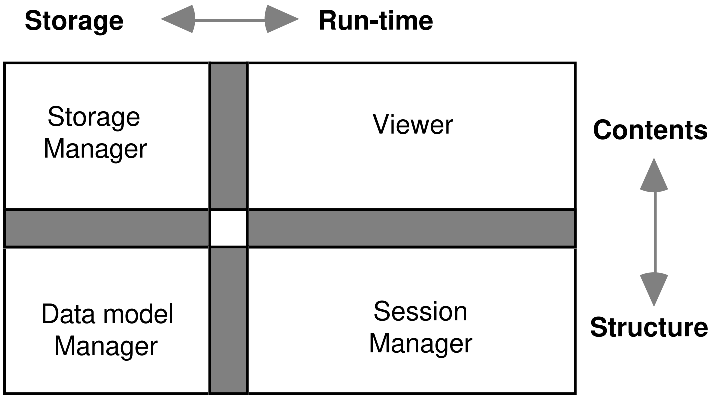


Figure 5: The Flag Taxonomy

What makes the flag taxonomy so appealing is that one can move the different configurations of the four functional modules and protocols and that most (if not all) configurations can be assigned a meaningful semantics corresponding with real systems. So the flag taxonomy enables a rough classification of hypermedia systems.

Such is depicted in figure 6, where a white region represents a unit in the hypermedia system that is a union of one or more functional modules in the flag. A greyed region stands for some protocol that exist between two modules and is accessible from outside. A black region represents a boundary between modules where no communication is allowed.

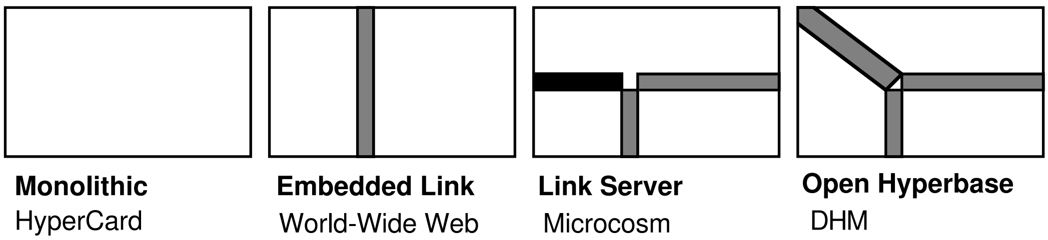


Figure 6: Some Meaningful Configurations for the Flag

A monolithic hypermedia system is a system where outside the hypermedia system one cannot distinguish the four functional modules (i.e. HyperCard) and is represented as a solid white rectangle. Embedded link systems like the plain vanilla world-wide web make a distinction between storage (i.e. a HTTP server) and run-time presentation (i.e. any web browser) but mixes structure with contents. Hence the vertical protocol between storage and run-time. A link server like Microcosm enforces third party applications to store and present their own data and as far as the hypermedia system concerns, there is no distinction between the storage and representation of data. However, third party applications can freely access the link services (horizontal grey protocol) using the linking protocol. Microcosm provides partial control over the storage and retrieval of links via the filter concept (vertical grey protocol) but this is not directly accessible for third party applications (horizontal black protocol). Open hyperbases have an extra service over link servers by offering viewer applications the possibility to store their data inside or outside the hypermedia system (the diagonal grey protocol).

The flag taxonomy can be used to separate open hypermedia systems from other hypermedia systems. The link server and open hyperbases are considered open hypermedia systems, as they do not impose a special data model on the viewer applications. The embedded link approach is not considered open, as viewer applications must adhere to the structure and contents format. According to the flag model, to be an open hypermedia system, there should exist a linking protocol between the session manager and viewer application. Note that the flag taxonomy represents the hypermedia system developer's point of view, which is opposed to the software system developer's point of view. In the latter, all systems with at least one protocol are open.

Although the flag taxonomy captures the essence of open hypermedia systems (i.e. separating storage from presentation and structure from contents) in an appealing metaphor, some important issues are not covered in the taxonomy. Only half of the requirements of the open hypermedia system definition are covered in the flag taxonomy; that is requirements 1 to 4 (size, data formats, applications) but not requirements 4 to 6 (data models, platforms and users). So the flag taxonomy covers only the three requirements that deal with interoperability, but not the ones that are about extensibility and distribution. However, one of the factors for adopting the requirement list was precisely that it covered the three main themes found in all open systems — interoperability, extensibility and distribution. So, we accept the design guidelines forwarded by the flag taxonomy (i.e. separate storage from presentation, separate contents from structure) but to study open hypermedia systems as a representative for other kinds of open systems, we need a more generic model. This more generic model is presented in the next section under the name of the Zypher perspective.

Conclusion

Open hypermedia is an active research area, so it is difficult to provide a precise picture of the field. To cope with this problem, we have presented three perspectives common in the field today.

The first perspective is presented as a scenario of an imaginary system for supporting the working practices of engineers participating in the design and manufacturing process of airplanes. The second —closely related— perspective is presented as a definition and an extensive requirement list. We have examined the items in the list, have sketched a number of issues and have pointed at common themes. An important observation for the remainder of the dissertation was that the requirement list covered important themes relevant in other kinds of open systems — i.e., interoperability, extensibility and distribution. This shows that open hypermedia is an excellent choice for experimenting with techniques to build open systems. The third perspective is presented as a taxonomy, which is a first attempt to summarise the important design principles for open hypermedia systems. The taxonomy forwards two essential separation of concerns in the design of open hypermedia systems, i.e. contents from structure and presentation from storage.

The Zypher Perspective

This section introduces our personal perspective on open hypermedia systems. We present this perspective because we feel that what is currently available lacks some important aspects on open hypermedia systems.

Reviewing our hypertext & hypermedia definition (p.08), we notice the importance of the word "style", which is about freedom, about customisability. Also, the open hypermedia definition (p.22) regards openness as the ability to incorporate new objects into as system, which is about adaptability. Although, style and adaptability are implicitly part of the requirement lists and scenarios, we want to make them an explicit element of our perspective since we want to focus on techniques to construct customisable and adaptable systems. Therefore, we present tailorability as a characteristic that encompasses both style and adaptability.

With respect to the flag taxonomy, we see the Zypher perspective as "another side of the coin". There are two arguments sustaining the position that the other side of this coin is indeed important. First of all, there is the way the flag taxonomy deals with the world-wide web. According to the flag taxonomy (see figure 6), the world-wide web is not an open hypermedia system because it violates the "get the links out of the data" principle. This may be true from a hypermedia system developer's point of view, but with its extensible architecture system developers would certainly classify it open. Secondly, there's is the way the flag taxonomy differentiates between Microcosm and DHM. Looking at figure 6, the flag taxonomy focuses on the way open hypermedia systems like Microcosm and DHM deal with the storage of documents. DHM provides hyperbase services, i.e. it allows third party applications to store the contents of the documents inside the DHM database, while Microcosm does not. We agree that this is an important difference, yet we see the difference between the componentware and framework approach (see "The Open Hypermedia Systems — DHM and Microcosm]" - p.15) at least as important. This difference is not covered in the flag taxonomy and yet it is important, also from a hypermedia system developer's point of view.

We propose tailorability as an encompassing view on openness. That is, we see tailorability as a way to achieve the kind of customisability needed to accomplish a true hypermedia style, but also as a way to capture the extensibility of the world-wide web, Microcosm or DHM. More importantly, we propose tailorability as a way to achieve openness in all kinds of open systems, such as operating systems, databases, inter application communication, tailorable software and programming languages. To support such an encompassing view, we need to be more precise on what is meant by the term tailorability. We define tailorability based on the notion of a design space as a generic abstraction of all kinds of open systems and identify three levels of tailorability afterwards.

The Zypher Design Space

Summarising the section on "hypertext & hypermedia" (p.07), we define a hypermedia system as a unique combination of navigation, storage and presentation technology. This summary is named the Zypher design space and it is shown in the left hand side of figure 7. The Zypher design space has three dimensions, known as the storage axis (enumerating all possible information repositories), the presentation axis (enumerating all possible viewer applications) and the navigation axis (enumerating all techniques to specify navigation relationships). A particular hypermedia system is then a relation (in the mathematical sense of the word) between points on these three axes and is represented by a 3-dimensional volume in the design space; for the sake of simplicity we represent such a volume as a cube. As announced earlier, an open hypermedia system is then a tailorable volume in that design space (see the right hand side of figure 7).

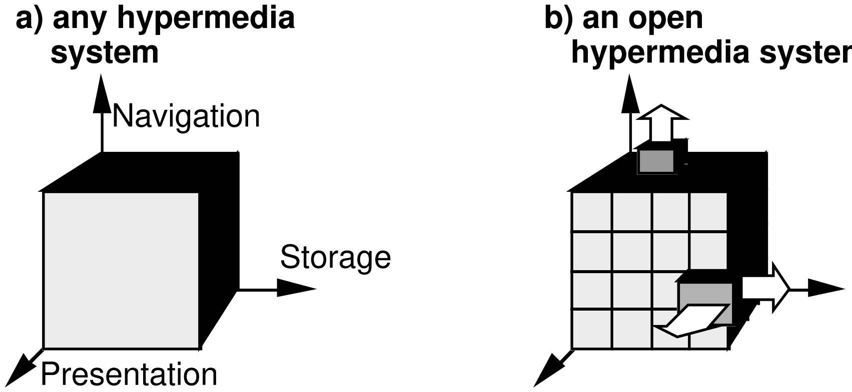


Figure 7: The Zypher Design Space

A very important remark is the choice of these axes is left open: the remainder of the argumentation is completely independent of the precise number and semantics of the axes. For our experiments we found those three valuable and we motivate their existence in the second part of this dissertation (see interoperability - p.78; navigation - p.86). However, a designer of an open hypermedia system —or any other open system for that matter— is allowed to change the axis system freely. The independence of the number and semantics of axes in the system is crucially important, as it is this aspect that allows us to generalise Zypher design space to any kind of other system. We come back on this issue when discussing the Zypher contribution (p.63).

Three Tailorability Levels

What is tailorability ? To a large degree, this question was answered during the discussion of the extensibility issue in the data model requirement (see OHS-REQ 4), where we list a number of techniques that can be used to adapt a hypermedia system (configuration tools, scripting languages & macro facilities, dedicated programming languages, general purpose programming languages, object-oriented frameworks, open implementations, extensible data models). This answer is not a satisfying one, as it emphasises on the techniques.

We propose a more analytical approach by presenting three levels of tailorability. Each level of tailorability is defined using the axes system of the Zypher design space in figure 7. Note the important property that *the definition of the three levels of tailorability is independent of the type and numbers of axes in the design space*.

Domain Level Tailorability

Domain level tailorability is the kind of tailorability needed to deliver a hypermedia system for a specific application domain. Typical usage of domain level tailorability in hypermedia systems would involve the incorporation of new data formats, extra information presentation methods or supplementary types of navigation techniques. *Domain level tailorability corresponds with the addition of extra points (or the modification of existing points) on the axes of the Zypher design space.*

System Level Tailorability

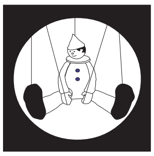
System level tailorability aims to deliver services that affect the global behaviour of the hypermedia system. Examples are concurrency control (i.e. manage concurrent access to shared data), logging (i.e. maintaining a log of navigation activities to provide backtracking features), caching (predict future behaviour on the basis of registered activities), authority control (check whether the user of the system has the privileges to see or modify information) and integrity control (control operations to preserve the consistency of the system's data structures). All these examples have in common that they change the behaviour of several modules in the system in a uniform way. *System level tailorability corresponds with wrapping uniform behaviour around different points residing on an axis of the Zypher design space.*

Configuration Level Tailorability

Configuration level tailorability aims to provide a 'plug and play' hypermedia system, where the coordination between the system modules is adapted without changing their internal implementation. Typical usage of configuration level tailorability is to make the system run-time extensible. For instance, the table of helper applications in most world-wide web browsers, where users can associate a viewer application with a given document type is about the configuration between the storage and presentation axis. The Java approach [Java'95] is another example, where world-wide web browsers load system modules for handling unknown document formats. The final example is protocol negotiation, where system modules communicate to choose the optimal protocol for exchanging data; protocol negotiation is valuable in cross-platform communication. *Configuration level tailorability corresponds with changing the relationship (in the mathematical sense of the word) between the points on the axes in the Zypher design space.*

The Puppet Master Metaphor

These three levels of tailorability play a crucial role in this dissertation. To make sure the reader recognises the differences and associates them with the appropriate level, we have devised the so-called puppet master metaphor. From this metaphor we have inferred three icons that are repeated throughout the text as little memory aids. The metaphor is based on the analogy between adapting a software system to a changing environment and adapting puppets to a new puppet play.



Domain Level Tailorability

When preparing a story, the puppet master conceives a number of puppets playing different characters. Archetypal puppet characters are the harlequin and the pierrot, where the former wears a costume with much coloured patchwork and a smiling face and the latter is dressed in white and holds a tear under the eye. To distinguish these characters the puppet master paints the puppet faces and dresses them up with different costumes.

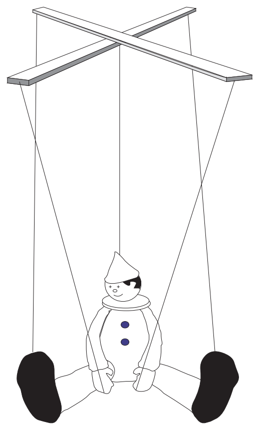
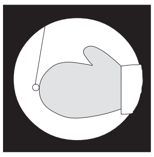
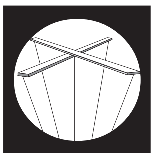


Figure 8: Puppet Master Metaphor



System Level Tailorability

However, for certain kinds of stories, some puppets require special abilities not captured by the basic puppet design. Take the example of a story that has a sword fighting scene. Pasting a sword onto the hands of an existing puppet is not enough, since a realistic fighting scene requires the ability to control the movements of the sword as well. Such a scene demands for an extra string to be attached to manipulate the special behaviour.



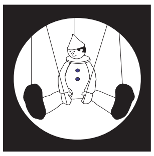
Configuration Level Tailorability

Puppet plays require the puppet master to co-ordinate the behaviour of each individual puppet. Some puppets need limbs that can be moved around freely, while other puppets require them to go in opposite directions. One must adapt the wooden cross co-ordinating the movements to achieve this kind of tailorability.

Note that the users of the system correspond with the audience watching the puppet: they are not supposed to know how the puppet is manipulated to produce the desired scenes in the play. However, just like the audience can influence the play by applauding and shouting, users can influence the behaviour of the system by setting preferences. The puppet master (i.e. the software engineer) then applies the tools on the appropriate level of tailorability to satisfy the audience.

Appropriateness of the Three Tailorability Levels

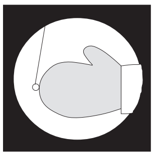
Having identified tailorability as a distinctive characteristic of open hypermedia systems and having proposed three levels of tailorability to handle different kinds of flexibility, the natural question that follows is how far these tailorability levels correspond with existing open hypermedia research ? To answer that question, we contrast two representative open hypermedia systems (Microcosm representing the componentware approach and DHM representing the framework approach) on each level and refer to the open hypermedia definition & requirements (p.22) to sustain our arguments.



Domain Level Tailorability

All open hypermedia systems satisfying the "size", "data formats" and "applications" requirements (see OHS-REQ 1, OHS-REQ 2 and OHS-REQ 3) are domain level tailorable, because satisfying these requirement boils down to lengthening the storage or presentation axes of the Zypher design space. For example, DHM has been tailored towards the domain of co-operative engineering (see [GrønbaekEtAl.'94]) by incorporating different viewer applications demanded by the engineers. Microcosm incorporated different multi-media viewers to deal with the Mountbatten and Churchill archives (see [FountainEtAl'90]). While discussing OHS-REQ 4 we have advocated the idea of an extensible link engine, which is essentially a stretching out of the navigation axis. Microcosm provides the generic link and filter concepts (see [Hill,Hall'94]) and DHM provides the notions of a location specifier and a reference specifier (see [Grønbaek,Trigg'96]).

Yet, we classify Microcosm more domain level tailorable than DHM, because Microcosm is more flexible. Extending the storage or presentation axis in Microcosm, requires the hypermedia system designer to tailor a viewer application to include calls to the Microcosm API and to perform some trivial installation procedure. With DHM a hypermedia designer would spend a comparable effort extending the viewer application, but installing it into the DHM framework is more difficult since it involves writing specialised subclasses.



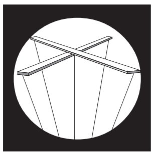
System Level Tailorability

Especially in collaborative and distributed settings (see OHS-REQ 5, OHS-REQ 6) the need for system level tailorability becomes apparent. To implement the concurrency control required in such settings, one needs to incorporate special locking or notification strategies. DHM relies on database technology and object-oriented inheritance (see [GrønbaekEtAl.'94]) to incorporate special locking strategies[7].

Microcosm —although extended for a distributed setting (see [Hill,Hall'94])— does not support different collaborative settings because there is no easy way to wrap additional locking or notification behaviour around system modules in a uniform way. This is largely due to the componentware approach taken by Microcosm, where it is very easy to customise the behaviour of one single module, yet very difficult to extend all modules with similar behaviour. In DHM it is more easy to incorporate uniform locking or notification behaviour because there one can rely on object-oriented inheritance mechanisms.

7 Hyperform ([Wiil,Legget'92] and [Wiil,Legget'93]) is an example of an open hypermedia system that took a similar —yet more flexible— approach by permitting dynamic subclassing of locking and notification classes.

The same argumentation can be repeated for other system uniform behaviour such as logging, caching, authority control and integrity control. In a framework approach this is relatively easy to do, because one can inherit such behaviour from a special class. In a componentware approach this is more difficult since there is no way to change the internal implementation of components uniformly.



Configuration Level Tailorability

DHM relies on object-oriented framework technology to support system reconfiguration. The hypermedia system designer has to provide some specialised classes and register them into the framework's session manager [Grønbaek,Malhotra'94] and the framework architecture ensures that they get called on the appropriate moment. Microcosm trusts on its extensible free-format message passing protocol to achieve reconfiguration [Davis,Knight,Hall'94]. The Microcosm link service passes a message to all appropriate modules which have to decide for themselves whether they want to react.

The difference we observe here is a difference between a centralised versus a decentralised approach. Frameworks in general, always have a tendency to centralise the system configuration making it easier to reuse a system configuration across different framework incarnations (see the discussion on object factories - p.54). Componentware in general has the tendency to distribute the system configuration over the participating components, making it easier to perform local reconfigurations. Both approaches have their advantages and disadvantages and the one approach can not be preferred over the other.

We conclude that the three levels of tailorability are appropriate for classifying open hypermedia systems. Also, these three levels of tailorability allow us to differentiate between the componentware approach (represented by Microcosm) and the framework approach (represented by DHM) for open hypermedia systems. The componentware approach is more domain level tailorable but less system level tailorable than the framework approach. As far as system level tailorability concerns, we see that both approaches are different and that there is no objective criterion to prefer one over another.

In fact the notions of domain level and system level tailorability reveal the inherent distinction between both approaches. Componentware is just about making it easy to tailor one component (i.e., one point on a design axis) so is always more domain level tailorable. On the other hand, the framework approach emphasises on reusing the same solution over a range of components (i.e., a complete design axis) so is more system level tailorable.

There is one concluding remark to make on the appropriateness of the three levels of tailorability. As follows from the above discussion, the three tailorability levels cover all the issues in the requirement list and thus the three central themes of open systems — interoperability, extensibility and distribution. Indeed, domain level and configuration level tailorability are primarily concerned with interoperability and extensibility, while system level tailorability covers aspects that have to do with distribution and sharing. *So we assert that the three levels of tailorability can be generalised to any kind of open system.*

A Framework Browser Scenario

Above we presented the Zypher design space and the three tailorability levels, which form the viewpoint that has been driving our experiments. To measure the strength of our viewpoint and to illustrate the scope of open hypermedia we envision, we pick up the idea of an open hypermedia usage scenario. The structure of our scenario comes from the sample published on the world-wide web at http://www.csdl.tamu.edu/ohs/; although we removed some subtitles we found less important and added one title "Our Results" explaining how far our experiments succeeded in achieving the goals of the scene.

Indeed, the scenario describes the vision that has been driving the design of the Zypher open hypermedia system and that has been partially realised in the many experiments we conducted. Nevertheless, the full completion of this vision has yet to be realised.

The scenario draws upon our experiences developing a Group Decision Support System (GDSS) [Kenis'95]. GDSS research is part of the larger computer supported co-operative work (CSCW) aiming to support the negotiation process in large groups and to improve the quality of group decisions. The emphasis of this research is on finding suitable communication structures that ameliorate the knowledge transfer between the participants, this way ensuring that more alternatives are considered with more wisdom. A computer may support diverse aspects of such a structured communication process, such as

- the identification of potential conflicts and possible compromises;
- the classification of different ideas, alternatives and opinions within the group;
- the shift from one decision making phase to another (support the 'chair' function);
- the publication of intermediate reports and final decisions (the 'secretary' function).

The scenario itself describes a framework browser as an integrated set of tools that supports an interdisciplinary team in the development of an object-oriented framework. This is another kind of engineering process than the Boeing counterpart presented in [Malcolm,Poltrock,Shuler'91]. We do not claim that this scenario represents the best approach to object-oriented software engineering, nor do we state that the scenario covers all aspects of the framework development process that should be supported by an imaginary framework browser. However, we are confident that the scenario represents a valuable approach towards software engineering and we use citations from [Goldberg,Rubin'95] to sustain our viewpoint.

The reader should be aware that object-oriented frameworks play several roles in this scenario. First, frameworks are the application domain for which we describe an imaginary open hypermedia system. In this sense, the term "framework" denotes all possible frameworks that a group of software engineers might conceive and it is represented by the GDSS framework in this scenario. Second, the scenario is about an integrated set of tools that support a framework development process. The integration is accomplished by means of an imaginary open hypermedia system with three levels of tailorability. In this sense, the term "framework (browser)" stands for tools and is represented by an imaginary framework browser in this scenario. Finally, the scenario illustrates the contributions of the Zypher experiments, which is a concrete incarnation of the above "framework browser" implemented using framework and open hypermedia technology. In this sense, the term "framework" denotes one particular framework providing a reusable design for the domain of open hypermedia frameworks and is named Zypher.

Especially when giving examples and discussing our experiments, it is hard to distinguish between those three roles, because in our experiments all roles are played by one and the same Zypher framework. Note that the Zypher framework has been implemented in the VisualWorks/Smalltalk environment and that the Zypher framework browser incorporates most of the programming tools that come with this environment, such as the system browser, class browser and method list browser. We refer to [Goldberg'84] for more information about these code browsers.

Scene 1: User Interface Prototype

Dirk is a sociologist and has been involved as a consultant in several projects supporting decision making in policy problems. Dirk applies a methodology based on the Delphi method comprising three cycles of paper questionnaires that are distributed among the group members by surface mail. Between each cycle, Dirk analyses the answers, rewrites the questionnaire and redistributes them among the group members. This process takes about a year to produce the final report and Dirk would like to speed it using computers. To illustrate his vision, Dirk uses HyperCard as a user interface prototyping tool. He authors cards for all the screens he believes are necessary in such a GDSS and links them together to create the effect of a working system.

Goal(s) of this Scene

"*Make the end user an insider. Successful teams enrol the end user as an insider, an active member of the product team.*" ([Goldberg,Rubin'95], p.294)

This scene stresses the importance of end user participation, already in the very early stages of software design. The scene shows that a framework development environment should support end users to prototype the user interface and control flow of their system. HyperCard represents the level of end user programming required in such an prototyping environment.

Character(s)

Dirk is the only character in this scene, playing the role of a requirements provider, aggressive tester and enthusiastic supporter ([Goldberg,Rubin'95], p.305). Dirk is a heavy user of all kinds of desktop publishing programs and is able to use such programs to express his ideas. He claims he does not know anything about writing programs, but he has experience with scripting languages and macro facilities that come with the desktop publishing software.

Framework Browser Requirements

To achieve true commitment, end users should play an active role and be able to build parts of the final program themselves. As end users typically do not have the technical expertise to program in full-fledged object-oriented software development environments, a componentware programming environment (like HyperCard) should be integrated with the framework browser. However, to achieve maximum effect, the components offered in the end user environment should correspond with concepts in the application domain. In our scenario, the prototyping environment should include components representing questions, answers and participants besides the generic ones like field, button, label, icon that come along with traditional componentware.

⇒ A framework browser should include a componentware programming environment with components tailored towards the target application domain (OHS-REQ 4: Data Models; Domain Level Tailorability).

Our Results

The Zypher experiments do not support the above requirement. However, we were actively engaged in the ApplFLab project, investigating how one could make user interface builders incrementally refinable ([SteyaertEtAl'94], [SteyaertEtAl'95]). This research starts from the assumption that user interface builders are essential tools for the development of modern applications, and argue that the state of the art of the field lacks a way of incorporating new user interface components. The ApplFLab environment can extend its set of user interface components using a special purpose meta-level interface. We plan to combine the ApplFLab environment with the Zypher framework browser in the future.

Scene 2: Analysis and Design

Having finished the GDSS prototype, Dirk starts to look for a group of people that can turn it into a working software system. He pays a visit to Patrick, a member of the computer science department of his university. Patrick happens to have a great deal of experience with object-oriented frameworks and he convinces Dirk to build a GDSS framework. Dirk fancies this idea, because he wants to try the future GDSS in both synchronous-proximate (i.e. all participants are working in the same room at the same time) and asynchronous-distributed (i.e. the participants are connected by a WAN-network and participate at different moments in time) settings and Patrick promises that he will be able to do both.

Patrick is well aware of potential pitfalls when building frameworks, certainly for a domain where he has no experience with. He decides that a "design by wandering around" method is probably the best way to cope with the uncertainties and tries a prototyping approach. To kick of, he starts to browse through the documentation Dirk provided, meets a few times to discuss GDSS ideas and gradually builds up an understanding of what Dirk had in his mind when building the user interface prototype. Patrick finally writes a set of analysis and design documents using his favourite tools (i.e. the Microsoft Word editor and a public domain OMT editor). Patrick uses an open hypermedia system to create numerous hypermedia links representing relationships between the OMT-diagrams and the analysis and design documents.

Goal(s) of this Scene

"*Tools. […] Much of the effort involved in analysis and design involves handling information with complex interdependencies that have to be created and maintained over long time periods. Tools must be available to handle the information management aspect of a (analysis and design) method.*" ([Goldberg,Rubin'95], p.346)

This scene illustrates that framework development is more than writing an object-oriented program and that a framework browser should support other related tasks as well. Moreover the scene emphasises that software engineers have their working habits and their favourite specialised tools, so that a framework browser must delegate specialised tasks to third party applications.

Character(s)

Patrick enters the scenario. Patrick is the framework designer ([Goldberg,Rubin'95], p.497), the person who determines the architecture for a generic set of parts that can be used to form GDSS applications. In this scene, Patrick designs an initial set of concrete parts to illustrate how the generic parts fit together to form a particular GDSS application.

Framework Browser Requirements

To support the wide range of tasks involved in framework development, the framework browser must be able to integrate any other application. However, the two example applications are quite different with respect to ease of integration. Microsoft Word is an example of a software package that incorporates interoperability standards (i.e. DDE and OLE) and is highly tailorable (i.e. the VisualBasic scripting language) and thus well suited for integration with the framework browser. The public domain OMT editor stands for all other applications lacking such facilities. With the growing support of operating systems, one can expect that more and more applications will adhere to interoperability standards, yet legacy applications will always be around. Techniques to deal with such legacy applications (i.e. like the "universal viewer" [Davis,Knight,Hall'94]) are essential.

⇒ A framework browser should be able to integrate any other application, irrespective of the degree of tailorability of such client application (OHS-REQ 3: Applications; Domain & Configuration Level Tailorability).

Our Results

We have integrated Microsoft Word as a third party document viewer. We used the macro facilities in Microsoft Word to accommodate for the necessary glue and implemented the inter application communication with DDE (Dynamic Document Exchange) under Windows.

The object-oriented implementation of the Zypher framework, makes it fairly easy to reuse the solution to communicate with other applications supporting DDE. Other interoperability standards (i.e. OLE, OpenDoc) could be incorporated if we want to.

However, the glue code in Microsoft Word is tedious to write, large in volume and hard to maintain. There is a definite need for some kind of reuse mechanism to support the incorporation of external application and this need is even stronger as far as communication with legacy applications concerns. We did not experiment with legacy applications due to a lack of resources. Standardisation efforts, and especially the idea of a dual protocol shim like described in [Davis,Lewis,Rizk'96], may answer the need for reusability.

Scene 3: Implementation

Patrick starts implementing a series of prototypes. As his knowledge of the GDSS domain grows, he sees that some of the assumptions in the analysis and design documents are incorrect and that others or missing. As is encouraged in incremental software development, he modifies the analysis and design to reflect his new insights.

To convince Dirk of the progress of the project, Patrick asks Koen to join the team and to develop a really attractive user interface based on the HyperCard prototype. The two of them form a real team now, working on the same artefact and need to synchronise their efforts. Patrick and Koen encounter two types of synchronisation problems.

The first problem is the maintenance of the relationships between evolving analysis, design and implementation. For instance, Patrick carefully designed that part of the GDSS framework that exchanges messages between participants, as this part must vary when building a GDSS-system for a synchronous-proximate or an asynchronous-distributed setting. Patrick included a link from the section in the design document to the places in the code of his prototype that implements this design. However, Koen has code in his prototype that implements the same functionality and Patrick wants the target of his link to include Koen's code as well. In fact, Patrick wants his link to include all current and future code that implements his design; he needs some kind of "smart" links that establish the navigation relationships dynamically based on specific knowledge about the framework architecture.

The second problem has to do with simultaneous access to shared data and what is called "tele-presence". Sometimes Patrick is editing the analysis and design documentation while Koen is reading it, so Koen and Patrick need to be aware that they are simultaneously working with the same document. The viewers on Patrick's and Koen's machines must show the document in a special colour to illustrate they work in a simultaneous access mode, and each time Patrick saves the document Koen must see the latest version. Of course, this schema must work with all applications used to view and edit parts of the framework.

Goal(s) of this Scene

"*Framework Goal. Set up a development environment for all team members that enables the team to create, maintain and deliver applications.*" ([Goldberg,Rubin'95], p.376)

This scene portrays that incremental development and teamwork implies facilities for consistency maintenance and integrity control. Professional software development environments may include such facilities, but when a framework browser incorporates third party applications it cannot rely on these facilities.

Character(s)

Koen comes on stage now, playing the role of an implementor in an application production team ([Goldberg,Rubin'95], p.282), specialised in user interface design. Koen stands for other characters with other specialisation's (i.e. databases, networks, statistics) as well. The point we want to make is that framework development is teamwork, so a framework browser must support working in teams.

Framework Browser Requirements

Traditional hypermedia systems use static point-to-point links to structure navigation relationships, which means that the author of a hypermedia system must maintain the connections between the sources and targets of the links manually. However, in environments where those relationships change often, static links become a maintenance nightmare. One way to deal with dynamic structures is to have some kind of smart links that use knowledge about the information to infer relationships. In the scene, Patrick would create a 'smart link' that uses internal data structures of the implementation environment to return all occurrences of the message implementing the design.

⇒ A framework browser should be able to incorporate smart links incorporating knowledge from inside the implementation environment (OHS-REQ 4: Data Models; Domain Level Tailorability).

To implement tele-presence, the system must ensure integrity between different viewers, simultaneously manipulating the same data. This implies that the framework browser must be able to monitor open, change and close events and pass them to other viewers. Those viewers are in principle unaware of each other and may or may not run on the same machine.

⇒ A framework browser should be able to trap important events and notify other viewer applications, irrespective of the type or number of viewers involved (OHS-REQ 6: Users; System Level Tailorability).

Our Results

Zypher is equipped with a fully extensible link engine (see "resolver" - p.94) and we have plugged in algorithms for 'smart links' between analysis and design documentation on the one hand and the implementation on the other hand. Analysis and design documents may include HTML style anchors[8] with a special "browse:" keyword. This keyword identifies a smart link referring to an element of the implementation. Possible references are "system" (to open the global system browser), "class X" (to open a class browser on the class X), "class X method Y" (to open a method browser on the method Y in class X), "sendersOf X" (to browse all methods that send the message X), "implementorsOf X" (to browse all methods that implement the message X).

Zypher does not include any facilities for concurrent users, so we could not yet demonstrate tele-presence. However, using the meta-object protocol we can trap all open, close and change events (see "meta-objects" - p.116) and we implemented a layered event passing mechanism on top of it (see "events" - p.107). Using this mechanism, we have been able to synchronise two home cooked viewers (i.e. HTML and text) that are completely unaware of each other. When we save a HTML representation in a certain text file using the simple text viewer, the HTML viewer opened on the same text file updates its contents automatically. The same scheme could have been used to synchronise the contents of two third party applications running on the same machine (i.e. the Notepad and Microsoft Word) viewing the same file. Synchronising viewer applications running on different machines is more difficult but would be feasible as well: besides the obvious need for a high level communication

---

8 In the Zypher experiments, we use embedded links to avoid the editing problem (see OHS-REQ 1).

channel between the two machines, it would involve an extension of the meta-object protocol to pass events over distributed machines.

Scene 4: Reuse Library

After a few analysis/design/implementation iterations, Patrick and Koen succeed in building two GDSS's; one for a synchronous-proximate setting and one for an asynchronous-distributed setting. Dirk is now testing the software with a number of groups and is quite pleased with how it works.

Patrick and Koen actually build a reusable design for the GDSS domain. However, there solution probably contains parts that may be reused in other CSCW frameworks. They hand over all the analysis, design and implementation to Roel, who is responsible for the identification of reusable assets and for storing them in the computer science department reuse library. To maximise access, this library is accessible via the world-wide web.

Roel studies the received material and via discussions with Patrick and Koen he identifies those parts of the analysis, design and implementation that may be reused in other frameworks. For each reusable asset, he writes a design pattern, including an analysis section (based on some ideas in the original analysis), a contract section (based on the original design) and an implementation section (including references to the original implementation).

Roel is now ready to store the new design patterns in the library. As far as the design pattern document concerns, this is not so difficult: he relies on the multitude of HTML converters available. The references to the implementation impose a special problem however. The tools supplied with typical programming environments provide a narrow, class-oriented perspectives on class- and object structures. Design patterns are usually spread over several classes and only a few methods matter to understand one particular design pattern. Roel concludes that the tools provided with the programming environment are not suited as viewers for the implementation of design patterns. Roel decides to build special purpose viewers himself, using the tools provided by the open hypermedia system (i.e. a set of components to assemble design pattern views). Moreover, he wants to translate those views into HTML forms, so that the reuse library would accessible through the web.

The design pattern library gets larger and larger, and Roel encounters consistency maintenance problems. A design pattern does not work alone, but is supposed to be used in combination with others. However, it is quite difficult to manage links between design patterns that are known to be working together. Roel decides to use the extensible link engine of the open hypermedia system to plug in special algorithms that help in managing the consistency of the library. The idea is that if design patterns work together, the implementations of both design patterns will overlap to some extent. Roel extends the hypermedia system so that when a new design pattern is added, its contents is scanned for all references to the implementation and those references are stored inside a special database. This database is used to infer relationships between design patterns that use the same implementation.

Goal(s) of this Scene

"*Framework Goal. Set up a structure in which to plan and manage the process of acquiring, distributing, and maintaining reusable assets throughout the organisation.*" ([Goldberg,Rubin'95], p.223)

This scene illustrates the different approaches towards reuse within one framework and reuse across frameworks. To accomplish the latter, some kind of reuse library must be set up. There again —certainly with the advent of the world-wide web— an open hypermedia system can play an important role to make the library easy accessible.

Character(s)

Roel is the so-called reuse librarian ([Goldberg,Rubin'95], p.500), responsible for certifying, classifying, and storing new reusable assets in the corporate or project library. Much of the reuse assets in the library are based on the design pattern catalogue published in [GammaEtAl'93] although Roel collected many domain specific design patterns as well.

Framework Browser Requirements

Being able to call upon services of third party applications to view information is one of the selling arguments for an open hypermedia system. Yet, third party viewer applications are not always able to show the desired information, partly because this information is stored within the link structures managed by the hypermedia system. So an open hypermedia system must provide facilities that allow the hypermedia author to build its own viewer applications easily, probably based on some kind of componentware approach.

⇒ A framework browser needs powerful, easy-to-use programmable tools for building specialised viewer applications (OHS-REQ 2: Data Formats, OHS-REQ 4: Data Models; Domain & Configuration Level Tailorability).

The world-wide web is reaching the status of a universal exchange medium. An open hypermedia system should make it easy to exchange information with the world-wide web.

⇒ A framework browser must be able to exchange information over the world-wide web (OHS-REQ 2: Data Formats; Domain & Configuration Level Tailorability).

To support a library function in a framework environment, an open hypermedia system needs tools to link a new library item with the items already in the library.

⇒ A framework browser should be able to incorporate smart links to support consistency maintenance in large reuse libraries (OHS-REQ 1: Size, OHS-REQ 4: Data Models; Domain Level Tailorability).

Our Results

Zypher comes with a library of components that can be assembled programmatically to form new code browsers. We plan to integrate these components with the ApplFLab environment to attain a componentware approach (i.e. similar to [Wuyts'95]). Using VisualWave (a Smalltalk tool for generating HTML forms out of Smalltalk window specifications) we can publish these code browsers on the web quite easily.

To support the consistency maintenance of the design pattern library, we have extended Zypher's link engine to incorporate knowledge about design patterns. When the librarian stores a new design pattern in the library, he must provide a list of all the messages participating in the corresponding framework contract. All these messages are stored in a special database and Zypher's link engine queries this database to generate a pop-up menu of all related patterns.

Conclusion

We have presented our perspective on open hypermedia, which forwards tailorability as an distinctive property of open hypermedia systems, encompassing both the style and adaptability aspects of open hypermedia. We have identified three levels of tailorability and have defined them based on the notion of a design space.

A design space represents a range of systems in an n-dimensional axes system, where each axis represents a fundamental characteristic present in all systems but differs depending on the exact system one is considering. We propose three axes to model a hypermedia design space: a storage axis enumerating all possible information repositories, a presentation axis enumerating all possible viewer applications and a navigation axis enumerating all techniques to specify navigation relationships.

The three levels of tailorability are defined based on the adaptability of the axes system. We identify domain level tailorability which corresponds with the addition of extra points to an

axis; system level tailorability, which corresponds with wrapping uniform behaviour around points on one axis; and configuration level tailorability, which corresponds with changing the relationship (in the mathematical sense of the word) between points on the design space axis. The puppet-master metaphor is presented as a memory aid.
An important property of these three levels of tailorability is that they can be generalised for any kind of open system. This follows from the observation that the three levels of tailorability cover the important themes in open systems: interoperability, extensibility and distribution.

We have shown that the proposed perspective is appropriate for classifying open hypermedia systems by contrasting two representative systems Microcosm (representing the componentware approach) and DHM (representing the framework approach). We have concluded that the componentware approach is more domain level tailorable but less system tailorable. As far as configuration level tailorability concerns, we have observed a difference but we are unable to prefer one approach over the other.

We have illustrated the value of the proposed perspective with a scenario of a framework browser. A framework browser is defined as an integrated set of tools that support the development of object-oriented frameworks, and we have used the framework browser scenario to relate the three levels of tailorability with the requirement list part of the open hypermedia definition. This scenario describes the vision that has been driving the design of the Zypher open hypermedia system and that has been partially realised in the many experiments we conducted. Nevertheless, the full completion of this vision has yet to be realised.

CHAPTER **2**

Object-Oriented Software Engineering

Object-oriented programming —with its improved programming productivity, enhanced modelling power and better program maintenance— is among the most promising subfields of software engineering. This chapter investigates to what extent the concepts and techniques proposed by the object-oriented community provide solutions for tackling the tailorability of open hypermedia problems.

Incremental Development

What is Software Engineering ?

According to [Brooks'87] software systems are more difficult to build than any other kind of systems built in any other engineering discipline because their inherent complexity, non-conformity, changeability and invisibility. Software engineering is the discipline that investigates techniques to deal with those inherent difficulties of manufacturing software.

incremental development, is generally regarded as one way to cope with the inherent difficulties. Rather than regarding a software system as a static device that can not be changed once it left the design table, incremental development views a software system as a flexible LEGO construction. This way, running software systems can be easily adapted to the unpredictable evolution of their environment.

Object-oriented software engineering is the subfield of software engineering that has taken this incremental development approach as one of its main subjects of study.

What is Object-Oriented ?

It is quite conventional to answer the question "What is Object-Oriented ?" in a bottom-up manner, explaining basic object-oriented concepts (i.e. objects, messages, classes, inheritance and composition) and how they fit together. The problem with bottom-up answers is that they depend heavily on the subject the question is referring to. For example, the answer to the question "What is an object-oriented programming language ?" is quite different from the answer to the question "What is an object-oriented user-interface ?" (or a database, or a drawing package or a hypermedia system for that matter).

With the enormous amount of material that has been published on object-oriented software engineering, we assume the reader is familiar with the subject. So we tackle the question from a slightly different perspective and reformulate the question into "What does it mean to be object-oriented ?". Answering the reformulated question leads almost naturally the proclaimed benefits of object-oriented software engineering.

So what does it mean to be object-oriented ? According to [Goldberg,Rubin'95], p.46 "*something is object-oriented if it can be extended by composition of existing parts or by refinement of behaviours. Changes in the original parts propagate, so that compositions and refinements that reuse these parts change appropriately*".

This definition can be applied on the available software construction tools. A language is object-oriented if it allows to express composition of existing parts, refinement of behaviours and propagation of changes. Thus, encapsulation, inheritance and polymorphism make programming language object-oriented. Likewise, a database is object-oriented if it supports the storage of object models, i.e. data models that support encapsulation, inheritance and polymorphism.

But the definition can also be applied on running software systems. A drawing package is object-oriented if it supports the composition of existing parts (i.e. grouping), the refinement of behaviours (i.e. dragging or resising depends on the object one is manipulating) and propagation of changes (i.e. attributes like colour or size propagate through components of a composite). A user interface is object-oriented if allows to express the composition of existing parts (i.e. folders in desktop environments) and refinement of behaviours (i.e. highlighting behaviour changes according to the target user interface element) and propagation of changes (i.e. moving folders move their contents).

This definition is appealing because the composibility, refinement and propagation requirements map very well onto the incremental development approach. In the object-oriented community, different techniques have been forwarded to satisfy these requirements. Object-oriented frameworks and meta-object protocols are two of the more prominent techniques.

Object-Oriented Frameworks

A popular way to look at the progress in software engineering is to see every step as providing higher level abstractions for larger entities and techniques. Assembly language instructions were assembled into control structures, control structures were gathered into procedures, procedures were gathered into abstract data types, and with the advent of object-oriented programming we know how to gather abstract data types into inheritance hierarchies. Object-oriented frameworks are probably the next step in this evolution, where we learn how to gather inheritance hierarchies into reusable designs.

Object-Oriented Frameworks

a) Definition

An object-oriented framework is a state-of-the art technique in object-oriented software engineering. The field has reached consensus over the following definition: "*A framework consists of a set of co-operating classes and objects that make up a reusable design for a specific problem domain*" [Johnson,Foote'88], [Wirfs-Brock,Johnson'90], [Johnson,Russo'91], [GammaEtAl'95], [Goldberg,Rubin'95], [Cotter,Potel'95].

b) Categories of Frameworks

Many examples of object-oriented frameworks can be found in the literature. They can be classified in three categories.

Application Frameworks

These kind of frameworks make up a reusable design for the construction of applications with a graphical user-interface. Some well-known examples are Apple's MacApp [Schmucker'86], Borland's OWL, ET++ [WeinandEtAl'88] and Parcplace's VisualWorks.

Domain Specific Frameworks

These kind of frameworks make up a reusable design for a highly specialised domain. Compared to application frameworks that aim at horizontal markets, domain specific frameworks aim at vertical markets. Examples are Choices [CampbellEtAl'93] for the domain of operating systems, ET++SwapsManager [Birrer,Eggenschwiler'93] for the domain of financial engineering and DHM [Grønbaek'94] for the domain of open hypermedia systems.

Support Frameworks

Support Frameworks are similar to application frameworks but aim to support functionalities in between the operating system and the application (i.e. peripheral devices, databases, …). A prominent example is Taligent's CommonPoint [Cotter,Potel'95].

c) Object-Oriented Frameworks versus Class Libraries

According to the definition, frameworks are about reusing objects and classes, so one could confuse them with object-oriented code libraries. The crucial distinction between a framework and a class library is the idea of a reusable design. We elaborate on this distinction by adopting a list of differences from [Cotter,Potel'95] (p.57).

Behaviour versus Contracts

Class libraries are essentially collections of behaviours that can be called when needed. A framework provides collections of behaviours as well, but the essential add-ons are the framework contracts: the set of rules that governs the ways in which behaviours can be combined. Among others, the framework contracts include rules about what the framework provides, what the caller of the framework must provide, what is allowed in subclasses, etc. We elaborate on this notion of framework contracts in a later section (see framework contract, p.51).

Don't Call Us, We'll Call You (The Hollywood Principle)

With a class library, client programmers instantiate classes to create library objects and call their methods afterwards. Most frameworks do it the other way round: client programs pass their objects (in most cases these objects inherit from objects part of the framework) and it is the framework that calls the methods of the client objects. This way, the framework defines large parts of the control flow of an application and clients reuse it in their applications. To work properly, this scheme requires that clients supply legitimate objects, hence the necessity of framework contracts.

Implementation versus Design

The emphasis of class libraries lies on the reuse of implementation or code reuse, while the emphasis of frameworks lies on the reuse of design. Frameworks represent a generic design solution for a variety of problems called the domain (or application domain, problem domain). A framework is not a single program, but embodies a family of related programs and the common factor between these related programs is the framework's design expressed in the framework contracts. Each program in the family of related programs adapts this design to a specific problem, hence why frameworks are often called semi-finished applications.

Implementing Frameworks

a) Abstract Classes and Template Methods

A framework is a reusable design for a specific problem domain. The reusable design is specified with number of semi-finished inheritance hierarchies, constructed using so-called abstract classes and template methods, which are defined in terms of abstract and concrete methods.

An abstract method is a method without an implementation and an abstract class is a class defining at least one abstract method. Abstract classes are not allowed to have instances, since it is an error to call methods without an implementation. A method that overrides an abstract method by supplying an implementation is called a concrete method. A concrete class is a subclass of an abstract class that has overridden all of the abstract methods in its superclass chain. As opposed to abstract classes, concrete classes may be instantiated. The roots of inheritance hierarchies are usually abstract classes, while the leafs are usually concrete classes. Note that it is a legal operation to create an abstract subclass of a concrete class, so in the middle of inheritance hierarchies one may find both abstract and concrete classes.

A template method is a method with an implementation that calls abstract methods or other template methods. As such, a template method forms some kind of a skeleton program, where the abstract methods must be replaced by the appropriate concrete methods to make it complete. This 'skeleton program' idea is the way a framework specifies aspects of the reusable design and they are defined by means of template methods.

To make the definition of a abstract classes and template method concrete, we work out a (partial) framework for the domain of direct manipulation user interfaces. Such a framework includes a design for modelling the behaviour of sensitive regions like buttons and window close-boxes (see figure 9). When the mouse button is pushed outside the region nothing happens (1). If the mouse button is pushed while the cursor is within the region, some kind of highlighting is performed to give the user feedback (2). As long as the mouse button is down, the sensitive region adapts its highlighting depending on whether the mouse enters or exits the region (3). Once the mouse-button is released, the highlighting is undone and some action is performed depending on whether the cursor was inside or outside the sensitive region (4).

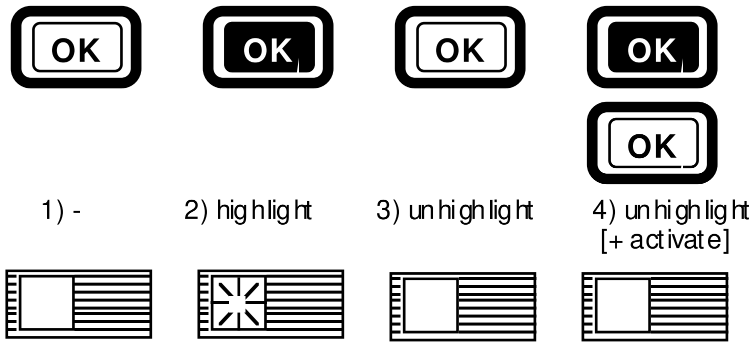
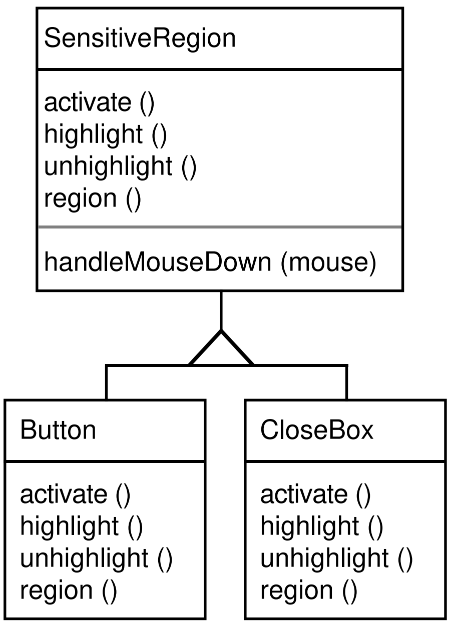


Figure 9: Sensitive Regions in a User-Interface

There are several ways to build a framework for a sensitive region and one way is presented in figure 10. There we have an abstract class `SensitiveRegion` with abstract methods `activate`, `highlight`, `unhighlight` and `region`. There is also a template method `handleMouseDown` that calls these abstract methods to model the mouse tracking behaviour. Subclasses of the sensitive region must provide different concrete methods to model their specific kind of highlighting and we included a subclass for a button `Button` and one for the close box of a window `CloseBox`.




```
method: handleMouseDown (mouse)
if contains(region(self),
            position(mouse))
  highlight(self)
  while buttonPressed(mouse)
    if contains(region(self),
                position(mouse))
      highlight (self)
    else
      unhighlight (self)
    endif
  endwhile
  unhighlight (self)
  if contains(region(self),
              position(mouse))
      activate (self)
  endif
endif
```

Figure 10: The `handleMouseDown` Template Method

b) White-box & Black-Box Reuse (Class Inheritance & Object Composition)

Frameworks are semi-finished programs: the idea is that clients reuse the framework design by filling in the open ends (i.e. supply concrete methods) of the framework. The two most common techniques to supply these methods are class inheritance and object composition but in terms of framework extensions the techniques are usually referred to as white-box reuse and black-box reuse [GammaEtAl'95].

For white-box reuse one relies on class inheritance within object-oriented programs. With white-box reuse a template method calls methods defined on the self object only; to extend the framework one supplies a concrete subclass of the class that implements the template method and overrides the appropriate abstract methods. The `handleMouseDown` method is an example of such a template method that is completed in the subclasses `Button` and `CloseBox` figure 10 by providing concrete implementations for the methods `activate`, `highlight`, `unhighlight` and `region`.

For black-box reuse one relies on the composition of objects. The template methods calls abstract methods defined in other classes and the concrete methods are provided by subclasses that are outside the inheritance hierarchy of the template method. In the example the `handleMouseDown` template method (see figure 10) calls the `contains` method, which is an abstract method defined on the class `Region` (not depicted). The `handleMouseDown` template method is reused by providing concrete implementations for the contains method in concrete subclasses like `Square` for the `CloseBox` and `RoundedCornerRectangle` for the `Button`.

The names white-box reuse and black-box reuse refer to the visibility of the implementation. White-box reuse implies subclasses that can (and must) look inside the implementation of the class defining the template method. With black-box reuse, the internal details of the classes defining the template method are completely hidden from the places where the framework must be extended.

In the remainder of this dissertation we use the term *white-box template method* to refer to a template method that implements a white-box reuse style, i.e. a template method that calls an abstract method defined on the self object. The term *black-box template method* refers to a template method that implements a black-box reuse style, i.e. a template method that calls an abstract method defined on another object.

c) White-box versus Black-Box Reuse

We illustrate the benefits of both approaches with an example. Suppose we must extend our user interface framework so that sensitive regions react to keyboard events. A white-box reuse approach involves the definition of an extra template method (`handleKeyDown`) that polls the keyboard instead of the mouse. A black-box reuse approach involves the definition of a new abstract class (`Tracker`) with one abstract method `handle`. The `Tracker` class has subclasses for all possible input-devices that may control the sensitive region, in our example `MouseTracker` and `KeyboardTracker`. The concrete method `handle` defined on the `MouseTracker` class is a minor rewrite of the older `handleMouseDown` method and the concrete `handle` method in the `KeyboardTracker` class is similar to the newly defined `handleKeyDown` method (hence the grey arrows in figure 11).

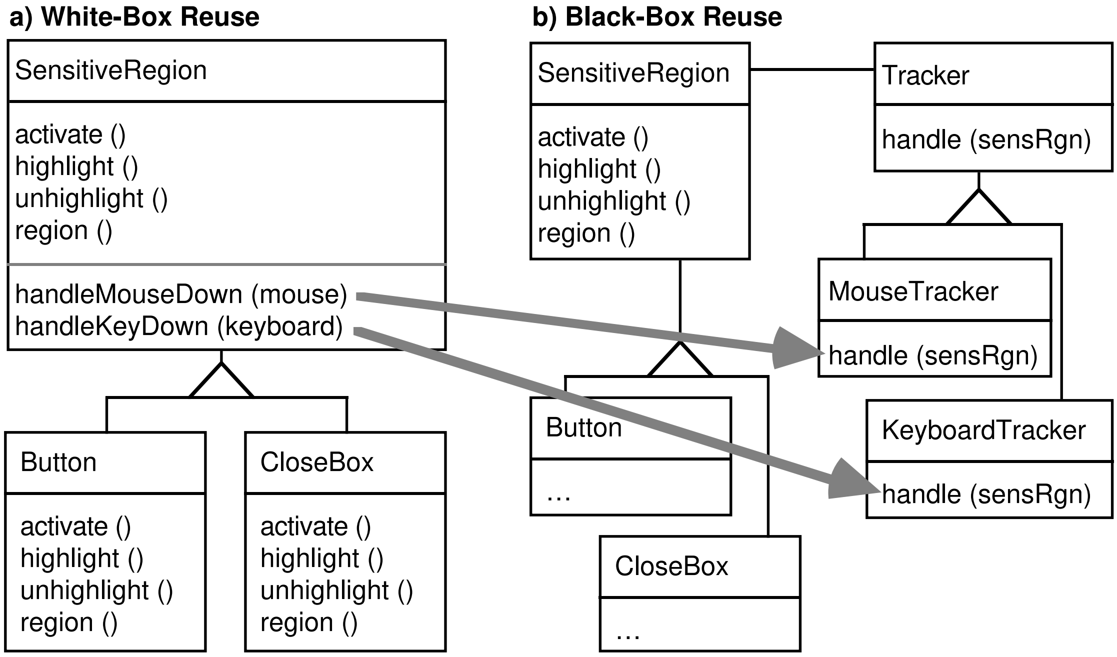


Figure 11: White-box versus Black-Box Reuse

The framework literature agrees that white-box reuse and black-box reuse both have their advantages and disadvantages [Johnson,Foote'88], [GammaEtAl'95], [Goldberg,Rubin'95].

Complexity

White-box reuse is more straightforward than black-box reuse because inheritance is better supported than composition. White-box reuse leads to a little number of large inheritance hierarchies which are usually easier to understand. Black-box reuse results in many smaller inheritance hierarchies with numerous and complex interrelationships in between them, thus difficult to understand. In the example, the white-box approach involves writing one additional method, while the black-box approach involves the creation of three additional classes and adjusting two methods.

Capability

White-box reuse offers more capabilities than black-box reuse because one can use internal implementation details when adapting the superclass. In the example, the window close-box and button may choose to override the `handleMouseDown` and exploit knowledge about the shape of the sensitive region to write a smarter polling algorithm. Indeed, the generic polling algorithm in the `handleMouseDown` method is written in such a way that it can handle disjoint regions by calling the `contains` function in each step of the polling loop. But a button and a close-box have a plain rectangular shape, and so the polling algorithm can be written faster by checking the distance the mouse has moved.

Encapsulation

White-box reuse breaks encapsulation because subclasses can look inside the implementation of the superclasses. Breaking encapsulation is sometimes necessary (see the capability argument), but is generally considered a bad thing as code tends to depend on internal implementation decisions and thus less extensible and harder to maintain. In the example, if the window close-box and button chose to override the `handleMouseDown` method then a change to the internal implementation of the basic `handleMouseDown` method in the `SensitiveRegion` class involves checking all subclasses to see whether they override that method.

Extensibility

For complex designs, black-box reuse usually involves less classes but more objects. This implies that black-box reuse is more extensible because classes are more task-specific and thus easier to subclass. Imagine that we must extend our user-interface framework to support other input devices like joy-sticks and three-button mice. The white-box approach requires us to add a new method to all of the `SensitiveRegion` subclasses for each new input-device that must be supported and there is little chance that we can reuse implementations of other input devices. In the black-box approach it is sufficient to add a new `Tracker` class, which is likely to be a subclass of an input-device with similar behaviour.

Configurability

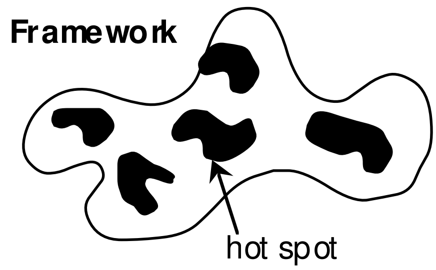
The inheritance chain of an object depends on its class and usually it cannot be changed at run-time. As white-box reuse depends on the inheritance hierarchy, one cannot change the reuse strategy at run-time. With black-box reuse, changing the association between the two objects is all that is required. Consider the example of a pop-up menu as a sensitive region where a user is asked to select an item out of a list. Some user interfaces allow the user to switch between the keyboard and the mouse while the menu is open. With black-box reuse, this can be modelled quite easily by associating two `Tracker` objects with a pop-up menu.

In general the advise is to favour black-box reuse over white-box reuse (see [Johnson,Foote'88] p.6; [GammaEtAl'95] p.20; [Goldberg,Rubin'95] p.208-209), especially since black-box reuse allows to reconfigure the system at run-time. However, black-box reuse distributes the design over a number of inheritance hierarchies and the system's behaviour depends on dynamic object relationships rather than that on static class relationships. This makes frameworks difficult to understand and difficult to build, hence the need for support.

Framework Development Techniques

a) Iterative Development

Figure 12: Framework & Hot Spots



A framework provides a skeleton application for a particular domain and uses abstract classes and template methods to anticipate those aspects of the domain that should be flexible. [Pree'94] coins the term 'hot-spot' to refer to an aspect of the domain that varies between applications and should be represented by an abstract class or method in the framework.

The inherent difficulty in developing frameworks is to identify the appropriate hot spots of the domain, because that is what makes up the reusable design. Framework researchers agree that finding these hot-spots requires domain-specific knowledge and iteration. The former is not a surprise, considering the domain-specific nature of frameworks; the latter needs a little more explanation as it is essential in the framework development process.

Developing a reusable framework requires that a framework be used and reworked several times to be confident in its quality and to make sure that the proposed hot spots correspond with reusable domain aspects. Iteration involves testing the framework in different applications and rework parts of the framework to remove mistakes and make improvements. Reworking the framework is sometimes called refactoring and shifts from white-box reuse to black-box reuse; i.e. tearing operations apart, moving methods up or down the hierarchy, rethinking interfaces, splitting classes and assembling others, …

b) The Agora, ApplFLab and Zypher Design Spaces

Domain knowledge is important to identify the hot-spots of a framework. In the PhD. dissertation [Steyaert'94], design spaces are identified as a particularly interesting way to look at problem domains with the intention to construct frameworks. A design space, captures the varying aspects of a problem domain in an n-dimensional axis system, where each axis represents a fundamental characteristic of all systems in the domain that varies depending on the exact system one is modelling. As such, a design axis corresponds with an important hot-spot of the problem domain.

Some experiments in our laboratory serve as a 'proof-of-concepts' for the design space abstraction. The Agora [Steyaert'94] design space captured the domain of object-oriented programming languages by proposing a two-dimensional design space with an object axis and a message-passing axis. The ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] design space captured the domain of user interface builders with a two-dimensional design space based on a widget axis and a value axis. The "Zypher design space" (p.30) captures the domain of hypermedia systems with a three-dimensional —storage, presentation and navigation— design space.

c) Framework Contracts

Frameworks are reusable designs and to reuse the design, clients must respect a set of rules telling how framework components must be used to work properly. Some of these rules can be inferred from the source code directly. For instance, abstract methods proclaim that framework clients should override that particular method. Types specify the minimal interface an object should respond to [Lucas,Steyaert,Mens'95]. However, frameworks favour black-box reuse and black-box reuse results in behaviour that depends more on dynamic object relationships than on static class relationships. This implies that most of the framework design rules cannot be inferred from the framework's source code, hence the need for additional techniques.

A contract [Helm,Holland,Gangopadhyay'90] is a mechanism for the explicit specification of dynamic object relationships and interactions. A contract defines a set of participants and their contractual obligations. The contract participants are objects with a well-defined interface. The contractual obligations state how contract participants must perform a sequence of actions and make certain conditions true in response to receiving messages part of their interface. A contract also defines invariant parts that contract participants should maintain. There exists a formal notation for contracts (see [Helm,Holland,Gangopadhyay'90]) but this is beyond the scope of this dissertation.

To illustrate the idea of a contract we give an example of the contractual obligations between the `SensitiveRegion` and `Tracker` abstract classes as introduced in figure 11. Note the rule that calls to the `highlight` method must be matched by a call to the `unhighlight` method, a constraint that never can be inferred from inspecting the framework source code.

SensitiveRegion

> A sensitive region object represents an area on the screen that is sensitive to certain input devices. Sensitive regions can be highlighted to give the user feedback. Sensitive regions have an action associated with them, that is executed when the input device activates the region.
>
> All sensitive region object implements an `activate` method sent by a tracker object to request the execution of the associated action. All sensitive region objects implement a `highlight` and `unhighlight` method sent by a tracker object to highlight (or undo the highlighting) of the sensitive region. All calls to the `highlight` method must be matched by a call to the `unhighlight` method.

Tracker

A tracker object represents an input device; input devices can activate sensitive regions on the screen.

All tracker objects implement a `handle` method; the caller of the method must provide a sensitive region object as parameter. The tracker objects calls the `highlight` method and then polls the input-device, repeatedly calling the `unhighlight` /`highlight` methods depending on the state of the input device it represents. At some point in time, the input-device must reach a state where it is finished working with the sensitive region, in which case the tracker object calls the `unhighlight` method. Depending on the state of the input-device when finishing, the tracker object may call the `activate` method to execute the action associated with the sensitive region.

d) Design Patterns

A frameworks is a reusable design and to reuse the design, clients must respect a set of rules known as the framework contracts. Yet, contracts are only a partial answer to the problems of framework development.

First of all, contracts do not tell why a certain rule is important, which hinders framework refactoring. Also, contracts fail to provide alternative designs, which hinders efficient reuse. Finally, contracts are very complex and difficult to write, so one wants to reduce the effort by reusing experience from other contracts in other frameworks dealing with similar problems.

Design patterns form an answer to these problems. Design patterns attempt to record the experience of expert framework designers by systematically naming, explaining, and evaluating important and recurring object-oriented design problems and their associated solutions [GammaEtAl'95]. A design pattern is a written document containing at least four elements: a name, a problem, a solution and list of consequences. The name is important to build a common design vocabulary in a group of framework developers. The problem section is necessary to understand when a design pattern is applicable; usually this part of the document contains some kind of analysis and an example. The solution section is a generic description of an object-oriented architecture that solves the problem. The consequences section include a discussion of the advantages and disadvantages of applying the pattern and motivates why the solution answers the problem.

Design patterns are collected in design pattern catalogues. Such catalogues make it easier to reuse a successful design and architecture, and are helpful for learning how to construct reusable designs. Moreover, a software engineer acquainted with a given design pattern applied in a new framework, encounters fewer problems understanding the framework and thus reusing it. Catalogues are useful for documenting frameworks as they build a common design vocabulary.

There are several attempts to assemble design pattern catalogues and they are classified in two categories. Domain specific catalogues collect patterns about a specific problem domain; HotDraw [Johnson'92], [Beck,Johnson'94] is a well-known example for the domain of graphical editors. Generic catalogues collect domain independent patterns: the most prominent example is [GammaEtAl'95]; [Pree'94] collects meta-patterns as all possible combinations for template methods.

The most appealing characteristic of a design pattern (i.e. a description of an object-oriented solution in an abstract, language independent way) is precisely its largest deficiency: design patterns are recorded using informal prose which makes them vulnerable for misinterpretation. Design patterns must be used in conjunction with other framework techniques such as framework contracts.

e) Object Factories

Experience has learned that it is very important to have flexible ways to create and compose objects within frameworks. This should not come as a surprise: frameworks favour black-box reuse over white-box reuse, implying that the heart of the framework is formed by loosely associated objects. Moreover, most frameworks adopt the Hollywood principle, i.e. the framework provides the general control flow of the application and client customise parts of it by supplying domain specific objects. Usually, the Hollywood principle assumes that clients supply their classes and that the framework creates the necessary objects. However, the framework itself cannot rely on client specific details and needs abstraction layers to hide direct references to client classes.

That is why most frameworks provide some kind of object factories (note that the generic design pattern catalogue [GammaEtAl'95] devotes an entire chapter to object factories). Object factories provide an abstraction layer that shields the framework from client specific details. Object factories are objects that have the special responsibility to create (and destroy) objects and object structures by means of some kind of abstract description. Object factories often involve a large body of the framework code and require special programming tricks, hence the need for supporting tools [VanLimberghen'96].

In the example of a user-interface framework, the design should be independent of particular subclasses clients provide for the `SensitiveRegion` class and the `Tracker` class. The user-interface framework may solve this by defining a special object that associates mnemonic names with specific classes. On start-up, the framework client installs the necessary classes in the association table and passes the mnemonic names as arguments to the framework operations that requests them. The user-interface framework calls the services of the object factory to create the objects, supplying the mnemonic name as a parameter. The user-interface framework may choose to incorporate a similar association table, to know which tracker objects can be associated with what sensitive regions.

Conclusion

Reuse is a way to reduce development effort. The object-oriented community proposes object-oriented frameworks as a technique to reuse a design for a particular problem domain. A framework is then a set of co-operating classes and objects, together with a set of rules on how they can be combined.

The set of rules is the crucial ingredient of a framework, and two important techniques exist that support the management of these rules. Contracts explicitly specify dynamic object relationships and interactions within a framework. Design patterns describe sets of rules that solve recurring design problems in an abstract, implementation independent way.

Frameworks are often called semi-finished programs in the sense that a framework consists of template methods that call methods without an implementation (the so-called abstract methods defined in abstract classes). Those template methods define the reusable aspects of the design. Clients must complete the framework by providing concrete classes with concrete methods overriding the abstract methods. As frameworks must be independent of the concrete classes provided by clients, most frameworks include some kind of object factories.

There are essentially two techniques for defining template methods, one is by class inheritance and the other is by object composition. Within frameworks, the techniques are usually referred to as white-box reuse and black-box reuse. The general advise is to favour black-box reuse over white-box reuse, because this makes frameworks more extensible and easier maintainable. However, black-box reuse is more difficult to understand and more difficult to build, hence the need for support.

The best support comes from an iterative development process, where a framework is tested in different applications and reworked to improve its design. A technique we found useful during experiments, was the specification of a design space. A design space, captures the varying aspects of a problem domain in an n-dimensional axis system, where each axis represents a fundamental characteristic of all systems in the framework domain that varies depending on the exact system one is modelling.

Meta-object Protocols

If progress in software engineering is about providing higher level abstractions, and if incremental development is about adaptability, then an obvious step is to provide abstraction levels that deal with the adaptation of a software system. That is the intent of an open implementation (often referred to as meta-level abstraction) — an abstraction technique that advances the state of the art in software engineering by providing abstraction levels that deal with the change of the system. Open implementations are a subject of study in the design of object-oriented implementations and there they are called meta-object protocols.

Open Implementation (Meta-Level Abstraction)

a) Black-box Abstraction versus Open Implementations

Software engineering is an engineering discipline and like all engineering disciplines, it is about coping with the complexity of the systems to build. The principal techniques for all engineering disciplines are abstraction and decomposition [Abelson,Sussman'84]. Software engineers have summarised these techniques into the principle of black-box abstraction, i.e. decompose a software system in modules where each module exposes its functionality through its interface but hides the way this functionality is implemented from the other modules.

Open implementation researchers claim that black-box abstraction is not always a good idea, because in many cases, clients want to influence the way the functionality is implemented. That is, if a module exposes only functionality, it hides away implementation issues that are not always details but often crucial strategy issues that bias the performance of the resulting application [Kiczales'94a], [Kiczales'94b].

b) Mapping Dilemmas, Mapping Decisions and Mapping Conflicts

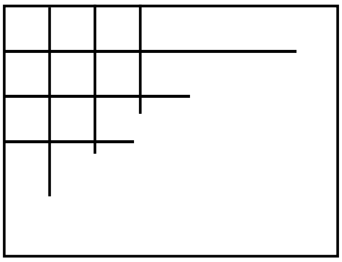
The claim follows from the insight that each black-box maps a higher level abstraction onto a lower level one. Often, such a mapping involves a mapping dilemma, i.e. a choice between several ways of mapping. The implementors of the black-box have to solve this dilemma taking a mapping decision, so the resulting implementation works better for some cases than others. However, particular clients may endure severe performance penalties by adopting the offered implementation and such a situation is called a mapping conflict.

The proposed solution is to open the box in a controlled way and allow the client to control the mapping decision hidden inside the implementation. Hence the name, open implementations.

c) The Spreadsheet Example

A recurring example in publications discussing open implementations (see [Rao'91], [Kiczales'94a], [Kiczales'94b]) is about implementing a spreadsheet on top of a windowing system.

When a windowing system is implemented as a black-box, it exposes functionality like sharing screen space and mouse-tracking and hides implementation details like window data structures and mouse-tracking algorithms. Having access to a functional interface for a windowing system, it is fairly easy to implement a spreadsheet (a rectangular array of non overlapping cells) by modelling each cell with a separate window, like shown in figure 13.



```
FOR i = 1 TO 100
   FOR j = 1 TO 100
      makeWindow (100, 100,
         i * 100, j * 100)
   END
END
```

Figure 13: The Spreadsheet Example of a Mapping Conflict

According to the principles of black-box abstraction this is a perfect solution. The engineer writing the spreadsheet application is not bothered with the underlying implementation details and can build a spreadsheet by a mere call to some `makeWindow` function for each cell in the spreadsheet. Doing this, the spreadsheet implementor would not need to write its own mouse-tracking algorithms and cell data structures.

However, the solution does not work in practice because the engineer writing the spreadsheet program is confronted with a mapping conflict. The mapping conflict is caused by the mapping dilemma involved in implementing a windowing system. Implementors have to choose data structures for the windows and algorithms for mouse-tracking and normally, they favour a solution that works well for a relatively small number of overlapping windows. Such a solution involves large window data structures and dumb mouse-polling algorithms. However, in the case of the spreadsheet the program would consume a huge amount of memory because the program reserves ten thousand memory segments each holding a complete window structure. Also, it would end up to be a very slow spreadsheet, because the mouse-tracking does not exploit the property that all cells are lined up one beside another but perform some sequential search through the list of all opened windows.

[Kiczales'94a] gives other examples (virtual memory, compilers) to sustain the viewpoint that mapping conflicts to some degree appear in all software modules that are reused by clients with different needs. This implies that mapping conflicts are more than just nice theoretical problems and thus that black-box abstraction should be extended to cope with mapping conflicts.

d) Base Level Interface and Meta-Level Interface

To deal with mapping dilemmas, open implementations propose a software engineer to control the mapping decision instead of hacking an entry into the black-box. However, open implementations do not want to loose the advantages of black-box abstraction and recognise the need to separate interface from implementation.

In open implementations, the separation of concerns is achieved by offering two separate interfaces, a base level interface and a meta-level interface. The base level interface is the functional interface that results from a traditional black-box design and exposes functionality only. The meta-level interface exposes implementation issues and is used to control the mapping decisions.

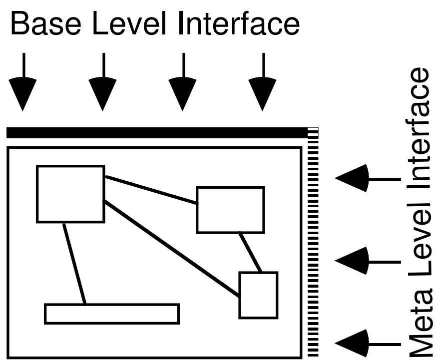


Figure 14: Base Level Interface & Meta-Level Interface

e) Designing a Meta-Level Interface

A well-designed meta-level interface is supposed to expose critical implementation issues and hide insignificant implementation details. Consider the example of a database system, where several techniques (i.e. hashing, B-trees, …) exist to tune the performance of the system. A software engineer tuning a database system is not forced to implement its own hashing table or a B-tree, but instead can fill-in some parameters to optimise the behaviour of the database to the needs of the application.

However, the question how to design meta-level interfaces for software systems is hard and remains open [Kiczales'94b]. The current thinking is that designing a meta-level interface is comparable in effort to the design of a base level interface and that similar techniques can be applied there [McAffer'95]. So abstraction and decomposition are necessary in the design of meta-level interfaces as well, hence iteration is crucial to filter out the significant issues that should be accessible through the meta-level interface. *However —and this is an essential problem of open implementations— iteration over the meta-level interface is more difficult and demands more resources than iteration over base level interfaces.*

To iterate over the meta-level interface, one must find a new application that requires a similar abstraction (i.e. base level interface) yet a different mapping decision (i.e. meta-level interface). If an application requires a different mapping decision, it means that the application tackles a different problem, thus that the application resides in a different application domain. To build applications for new application domains one must acquire new domain knowledge, a costly process as it involves new people, extra time and more money. Hence the conclusion that the design of meta-object protocols must be supported by theory and guidelines.

The contribution of this dissertation is a partial answer to the question how to design a meta-level interface (see the Zypher contribution - p.63). Before we can present the answer we need a little bit more background on the intuitive notion of a meta-level interface. We need to answer the question "what is meta ?", a question that is studied within the reflection community.

What is Meta ?

To summarise the previous section, an open implementation is an abstraction technique that proposes a scheme which allows client modules to negotiate with their service modules to find an optimum internal representation. Observing this scheme with a little more detail, reveals that a program with an open implementation somehow analyses its working and adapts its internal representation accordingly. This ability of self-analysis and self-adaptability is called reflection and has been studied within computer science for quite a few years now. With the advent of open implementations, reflection has found its way into software design.

An open implementation proclaims that a system should have two interfaces, a base level interface and a meta-level interface. To support the design of meta-level interfaces, we need an answer to the question "what is meta ?". The only problem is that there is no clear-cut answer to the question: reflection is an active field of research and there are exist many possible answers in the community.

Below we give an overview of two of the views that exist on the notion of reflection. And since there is no answer to the question "what is meta ?", we look for a criterion that helps us to decide "when is something meta ?".

a) Computational Reflection or Meta-Linguistic Abstraction

The term computational reflection stems from [Maes'87], while meta-linguistic abstraction comes from [Abelson,Sussman'84].

Computational reflection is a particular way to look at a software system. The view starts from the assumption that there exists a programming language used to express problems on a certain abstraction level. Expressions in that language are called programs and programs are turned into working systems (called computational systems) by means of an evaluator. An evaluator (called the meta-system) is a computational system that executes the program, i.e. it performs the actions required to realise the meaning of that program. The program, the computational system and the meta-system are depicted in the left hand side of figure 14.

Computational reflection is then the ability of a computational system to "*inspect and/or manipulate the representations of the computational process specified by a system's program*" [Rao'91] and is depicted by the grey arrow in the right hand side of figure 14.

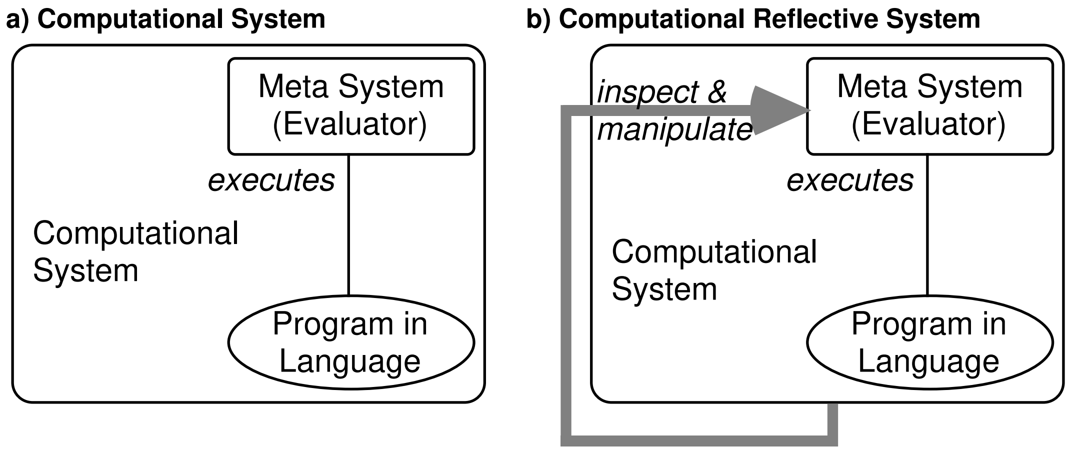


Figure 15: Computational Reflection

A well-known experiment applying computational reflection on program language design was the CLOS meta-object protocol [Kiczales,Rivières,Bobrow'91]. The experiment showed that reflective programming languages provide an extra abstraction layer, which eases extensibility, compatibility and efficiency. Reflective programming languages ease extensibility because it is possible to define a small and fixed kernel language and use that kernel to extend the language expressiveness; they ease compatibility because it is possible to ensure that earlier versions of the language lie within the scope of meta-extensions; they ease efficiency because one can differ the implementation strategy depending on the problem to program is solving. Moreover, programming environments for reflective programming languages may use the reflection abilities to monitor the evaluation of programs and construct powerful tools like debuggers and code optimisers more comfortably.

Computational reflection is a valuable view to look at reflection in programming languages. In [Rao'91], an attempt is made to adapt the computational reflection view to fit open implementations. This view is called implementational reflection.

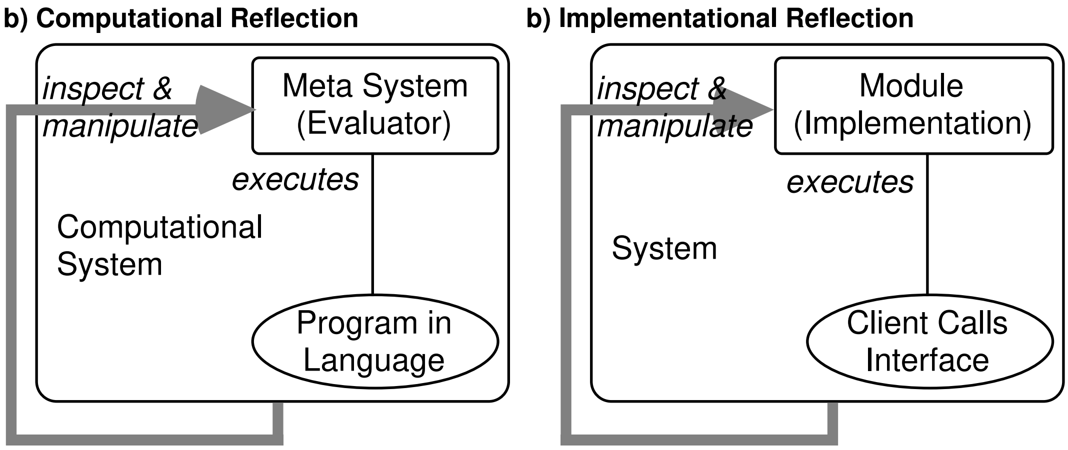


Figure 16: Implementational Reflection

The implementational reflection view is based on a one-to one mapping from programming language concepts to system concepts depicted in figure 16: a computational system is mapped on a generic system; the programming language is mapped on an interface of a module in the system; the evaluator is mapped on the internal implementation of that module and the program itself corresponds with system calls to the interface of that module. Implementational reflection is then the ability of a system to "*inspect and/or manipulate the implementational structures of the modules used by that system*" [Rao'91]. Seen this way, computational and implementational reflection are just different characterisations of the same essential capability, namely some form of self-analysis.

However, this mapping is not evident in systems that are not about programming languages. This follows from the fact that two co-operating subsystems use each others interface to communicate, so each of the two can be regarded as a meta-system for the other. Consider the example of a relational database system. Client systems send SQL queries to the database system, which evaluates these statements to produce desired data records. So in this case, the database system plays the role of a meta-system for the client system. On the other hand, client systems receive a stream of data records and must interpret that stream to identify the data records. So client systems play the role of a meta-system for the database.

We conclude that the implementational reflection is not able to tell which subsystem is the base level and which is the meta-level. However, this separation is essential for designing a meta-level interface (see p.57), and so the meta-linguistic view is not appropriate when discussing open implementations.

b) Definition of Reflection

Since the meta-linguistic view for reflection is not appropriate for open implementations, we have to search for another one. [Maes'87] proposes a definition for reflection that is applicable to any model of computation: "*A reflective system is a system which is about itself in a causally connected way*".

The three main ideas in this definition (i.e. system, about-ness and causal connection) need a little explanation to make things more precise. A 'system' is software running on a computer with the intention to answer questions about and/or support actions in some domain. A system incorporates internal structures representing its domain, that is why a system is said to be 'about' its domain. A system is said to be 'causally connected' to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect on the other.

[Maes'87] includes an example of a system steering a robot-arm to illustrate the definition. Such a system incorporates internal structures holding data values, relations and algorithms to represent the position of a robot-arm, so the system is 'about' the robot-arm position. These system structures are 'causally connected' to a particular robot-arm if (a) movements of the robot-arm originated by some external force changes the internal structures accordingly and (b) changes to the internal structure (by some computation) make the robot-arm move to the corresponding position.

When facing a software system with several subsystems, a rigorous application of this definition allows us to see what subsystems belong to the meta-level. A subsystem is part of the meta-level if and only if, that subsystem is (a) about the software system in (b) a causally connected way. If we reconsider the relational database example, we see that neither the database system nor the client system are meta-systems because they are not about the system but about the domain of the database applications. However, the parts of the database and client systems that exchange commands to create database structures (the DDL statements) form together the meta-system for the parts that exchange commands to query and update database structures (the so-called DML statements).

c) The Meta-Object Criterion

The last example makes clear that this definition of reflection is applicable as a criterion to judge whether a particular subsystem belongs to the meta-level, so is more appropriate in the study of meta-level architectures that are not about programming languages. So it is the latter view on reflection that is adopted for the remainder of this thesis.

When meta-level interfaces are studied within the context of an object-oriented implementation they are referred to as a meta-object protocol. Most open implementations are based on an object-oriented implementation, hence the reason why the terms meta-object protocol and open implementation are often used interchangeably. We rephrase our meta-level criterion in terms of objects.

Meta-Object Criterion

> *An object is part of the meta-level if and only if, that object is (a) about the software system in (b) a causally connected way.*
>
> In this criterion, 'being about X' means incorporating internal structures representing X and 'causally connected' means that if one of the sides of the connection changes the other side changes correspondingly.

c) Systems Meta-Object Protocols

The CLOS meta-object protocol [Kiczales,Rivières,Bobrow'91] is the most prominent example of an open implementation for an object-oriented language. Among others, this meta-object protocol has been used to make CLOS persistent [Paepcke'90].

Agora [SteyaertEtAl'93], [Steyaert'94], [CodenieEtAl'94], [Steyaert,DeMeuter'95] is a reflective, prototype-based language that features a general mixin-based approach to (multiple) inheritance. Agora uses its meta-object protocol to explore different object-oriented programming paradigms within a design-space of object-oriented programming paradigms.

CodA [McAffer'95] is especially important as it shows that a meta-level is "just another application" and that one can apply traditional software design techniques like abstraction and decomposition on the design of meta-level interfaces as well. CodA has been used to open up the implementation of Smalltalk message passing and is able to add meta-level infrastructure to Smalltalk objects so that additional behaviour like concurrency or distribution can be added when necessary.

Abstract Communication Types [AksitEtAl'93] is similar in aim but different in approach than CodA. Abstract communication types are objects that abstract interactions among objects and can be used to model distribution and concurrency. Unlike CodA, the abstract communication types are intended to be part of the design of a new programming language (i.e. Sina).

Apertos [Yokote,Teraoka,Tokoro'89], [Yokote'92] is an example of a meta-object protocol used in an object-oriented operating system for mobile computing. Because portable and mobile computers change locations frequently, the operating system must evolve and adapt itself to its execution environment. Apertos supports distribution and object migration.

SOM, the System Object Model part of IBM's OS/2 operating system [Forman,Danforth,Madduri'95] includes the notion of meta-classes, which can be used to wrap additional behaviour around base level message sends. Among others this can be useful in tackling persistency and concurrency.

Silica [Rao'91] is an open implementation for a windowing system. By providing special meta-objects, clients of the windowing system can fine-tune the Silica windowing system to their needs. In [Rao'91] it is described how Silica would handle the spreadsheet example described earlier.

ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] is an experiment that investigates what is needed to make a user interface builder incrementally refinable. ApplFLab starts from the assumption that user interface builders are essential tools for the development of modern applications, and argues that the state of the art of the field lacks a way of incorporating new user interface paradigms (e.g. direct manipulation, menu driven, navigational). ApplFLab is an object-oriented framework for a user interface builder and uses a meta-object protocol to install new interface paradigms.

Conclusion

To cope with complexity, software engineers apply a divide and conquer technique based on abstraction and decomposition. A system is decomposed in different modules, where each module exposes some functionality through an abstract interface, this way hiding its internal implementation. Each module maps high level functionality onto lower level functionality and this mapping restricts the possibilities of the higher level functionality.

An open implementation is a particular abstraction technique where the interface of a module is split in two separate parts: the base level interface and the meta-level interface. The base level interface is used to access the basic functionality of the module, while the meta-level interface allows to control the mapping imposed by the module. Experiments have shown that the open implementation abstraction technique makes systems more adaptable.

When meta-level interfaces are studied within the context of an object-oriented implementation they are referred to as a meta-object protocol. The separation of concerns principle dictates that it is better to have a clear separation between the base level interface and the meta-level interface. This implies that one needs a criterion to recognise meta-level objects in existing software systems. We adopted a criterion stating that an object is part of the meta-level if and only if, that object is about the software system in a causally connected way. In this criterion, 'being about X' means incorporating internal structures representing X and 'causally connected' means that if one of the sides of the connection changes the other side changes correspondingly.

The Zypher Contribution

Within today's software and hardware industry, there is a growing tendency towards openness. This tendency can be observed in areas like operating systems (e.g. UNIX and OS/2), databases (e.g. Exodus and CORBA), inter application communication (e.g. OLE and OpenDoc), tailorable software (e.g. Emacs and AutoCAD) and programming languages (Smalltalk and CLOS). More recently, the Taligent company [Cotter,Potel'95] —the joint venture between Apple, IBM and Hewlett-Packard— emerged as one of the most prominent advocates for openness, developing an open software platform that is able to keep pace with the growing demands and emerging technological innovations of the software market. Open hypermedia (p.22), is another example of a field where openness is considered a valuable, and —as argued earlier— the field is an excellent representative for the larger collection of open systems.

The software engineering research community has contributed to this open systems tendency by means of object-oriented frameworks and meta-object protocols discussed in the previous sections. Those techniques have found their way into the design of commercially available open systems: Taligent [Cotter,Potel'95] already identified frameworks as technological cornerstones for their CommonPoint open platform and IBM incorporated meta-objects in the System Object Model (SOM) underlying OS/2 [Forman,Danforth,Madduri'95]. Nevertheless, the today application of both techniques is more an art than a science and certainly the total effect of combining both techniques remains an open question.

We have performed the Zypher experiment to investigate the precise effect of these techniques by applying them on the design and implementation of an open hypermedia system. That is, we want to investigate why it is a good idea to use an object-oriented framework and a meta-object protocol in the design of an open hypermedia system and to what extent the two techniques overlap or differ. Moreover, it has been our explicit intention to be able to generalise our results to the design of open systems in other domains as well.

Three levels of tailorability: a Recapitulation

In the domain of software engineering, it is common practice to validate techniques experimentally [NationalAcademy'94] — researchers build prototype systems for a particular application domain to investigate benefits and drawbacks of a technique. Such a prototype system is called a software artefact and the Zypher open hypermedia framework is the software artefact we have been using for the experimental validation.

The design of the Zypher open hypermedia framework is based on the Zypher design space and the notion of tailorability as defined in the section entitled "The Zypher Perspective" (p.30). To develop our argumentation, we recapitulate the main ideas in what follows.

Open Hypermedia Working Definition

The Zypher design space is a three-dimensional (storage, presentation and navigation) axis system. Each dimension represents a fundamental characteristic of all hypermedia systems that varies depending on the exact system one is modelling. Each point on a design space axis is a particular variant of the fundamental characteristic.

A hypermedia system is a relation (in the mathematical sense of the word) between points on the three axes of the Zypher design space. An open hypermedia system is a "tailorable" hypermedia system.

Figure 17 represents a hypermedia system by a 3-dimensional volume in the Zypher design space; for the sake of simplicity we represent such a volume as a cube. An open hypermedia system is represented by a tailorable volume.

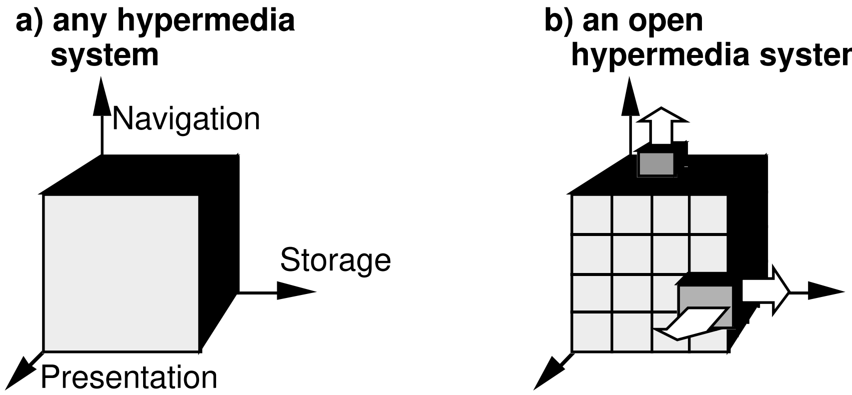
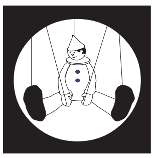


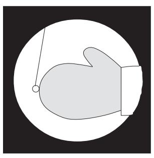
Figure 17: Zypher Design Space with an Open Hypermedia System

Our working definition proposes tailorability as the characteristic property of an open hypermedia system. We define three levels of tailorability based on three possibilities to tailor a volume in the open hypermedia design space.



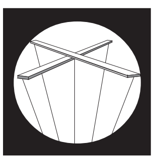
Domain Level Tailorability

Domain level tailorability is the kind of tailorability needed to deliver a hypermedia system for a specific application domain. Typical usage of domain level tailorability in hypermedia systems would involve the incorporation of new data formats, extra information presentation methods or supplementary types of navigation techniques. *Domain level tailorability corresponds with the addition of extra points (or the modification of existing points) on the axes of the Zypher design space.*



System Level Tailorability

System level tailorability aims to deliver services that affect the global behaviour of the hypermedia system. Examples are concurrency control (i.e. manage concurrent access to shared data), logging (i.e. maintaining a log of navigation activities to provide backtracking features), caching (predict future behaviour on the basis of registered activities), authority control (check whether the user of the system has the privileges to see or modify information) and integrity control (control operations to preserve the consistency of the system's data structures). All these examples have in common that they change the behaviour of several modules in the system in a uniform way. *System level tailorability corresponds with wrapping uniform behaviour around different points residing on an axis of the Zypher design space.*



Configuration Level Tailorability

Configuration level tailorability aims to provide a 'plug and play' hypermedia system, where the coordination between the system modules is adapted without changing their internal implementation. Typical usage of configuration level tailorability is to make the system run-time extensible. For instance, the table of helper applications in most world-wide web browsers, where users can associate a viewer application with a given document type is about the configuration between the storage and presentation axis. The Java approach [Java'95] is another example, where world-wide web browsers load system modules for handling unknown document formats. The final example is protocol negotiation, where system modules communicate to choose the optimal protocol for exchanging data; protocol negotiation is valuable in cross-platform communication. *Configuration level tailorability corresponds with changing the relationship (in the mathematical sense of the word) between the points on the axes in the Zypher design space.*

Note the important property that *the definition of the three levels of tailorability is independent of the type and numbers of axes in the design space*.

Tailorability and Object-oriented Software Engineering

Now that we have proposed three levels of tailorability as a distinctive characteristic of an open hypermedia system, it is time to reconsider the question how object-oriented frameworks and meta-object protocols can be applied in the design of an open hypermedia system. To answer this question, we look for a mapping between the three levels of tailorability and the two techniques provided by the object-oriented software engineering community. If we could find such a mapping, then we may conclude that the two techniques overlap since they both share the common goal of tailorability, yet differ in the exact level of tailorability that can be achieved.

We start with an intuitive mapping and elaborate on our ideas later.

Framework Level ⇒ Domain Level Tailorability

A framework provides a skeleton application for a particular domain and clients must supply concrete methods for the "hot spots" of the framework (i.e. iterative development - p.51). If we would design our framework so that each axis in the Zypher design space corresponds with a black-box template method (i.e. white-box & black-box reuse - p.48), then adding points to the design space axis corresponds with supplying concrete implementations for the abstract method in the template. So, a framework seems like a dedicated means to provide domain level configurability.

Meta-Level ⇒ System Level Tailorability

The crucial idea of system level tailorability is to adapt the global behaviour of the system in order to offer additional services. Locking is an example of such an additional service and to implement a successful locking strategy we must change all data-access operations in a uniform way: we cannot risk to miss a single read or write operation since this may violate data integrity.

We can accomplish locking behaviour by changing the implementation of all the modules in the system in a uniform way. This is not a very appealing solution however. First of all, we have to make sure that all modules that do call or implement a method that reads or writes a data-value are changed, but there are no tools to detect them. Secondly, changing the internal implementation of the modules implies that we must understand all internal details of each of these modules. This is hard, since read and write operations may be hidden deep inside the code. Finally, it is not always possible to change the internal implementation of a module, as it may be supplied by an external vendor.

Locking is a particular example, but the same is true for the other kind of services that depend on changing all operations of a certain kind. For instance, maintaining a trail of all visited locations implies that the system must log all navigation operations in a uniform way; to implement a cache, the system must trap all storage operations in a uniform way; authority control requires the system to trap all presentation operations in a uniform way; integrity control requires the system to trap all storage, navigation and presentation operations in a uniform way.

So to accomplish system level tailorability, we need a way to trap all operations of a certain kind, independent of the module that calls or implements that operation and wrap uniform behaviour around the operation. In other words, our hypermedia system must have a special interface that allows us to wrap extra code around these kind of operations. A meta-level interface seems like a dedicated means to provide this kind of functionality.

Meta-Level ⇒ Configuration Level Tailorability

Finally, configuration tailorability is about changing the configuration of the system's modules, without changing their internal implementation.

If we review the examples that come with configuration level tailorability, we see that the problem is in fact very similar to the system level tailorability problem: we need a way to change all decisions about the system's configuration in a uniform way. The minor difference is that for system level tailorability we must wrap additional behaviour, while configuration level tailorability requires us to change the behaviour. For instance, the table of helper applications is a decision table that tells which presentation module should handle the output of what storage module. The Java approach can be handled by checking the above decision table, and requesting the server to supply a presentation module if we do not find an entry. Protocol negotiation is an example on deciding which two modules should handle a certain service.

So to accomplish configuration level tailorability, we need a way to change all operations in the system that determine the system configuration in a uniform way. In other words, our hypermedia system must have a special interface that allows us to plug in special configuration code. A meta-level interface seems like a dedicated means to provide this functionality. Note that such a meta-level interface corresponds to the idea of an object factory (p.54).

Explicit Framework Contracts are Meta-Objects

So, it seems that a meta-object protocol is applicable to achieve both system level tailorability and configuration level tailorability. Our intuition has been guided by the insight that both levels of functionality could be incorporated by having two dedicated interfaces: one that allows us to wrap additional behaviour around important operations and another one that allows us to plug configuration code. Nevertheless, we have not motivated why such a dedicated interface is part of the meta-level.

In the section on meta-object protocols, we have identified the meta-object criterion (p.61) that enables us to decide whether an object is part of the meta-level. So if we know what objects are included in the dedicated interfaces, we should be able to tell whether they belong to the meta-level interface or not.

The encompassing property of the dedicated interfaces for system and configuration level tailorability, is that they capture the important design aspects of the framework. Indeed, the operations we want to log, cache, lock, control with system level tailorability are the operations that perform critical system tasks, so are important aspects of the framework design. Also, the operations that determine the system configuration are crucial for the behaviour of the framework, thus an important aspect of the system design. Important design aspects of a framework are normally specified using framework contracts, so we may conclude that objects part of the dedicated interfaces are explicit representations of framework contracts.

Using the meta-object criterion, we can show that *an object which is an explicit representation of a framework contract is part of the meta-level.* The meta-object criterion for an object to be part of the meta-level; that object should be (a) about the software system in (b) a causally connected way. Point (a) follows from the fact that an object which is an explicit representation of a framework contract is by definition about the system and not about the system's domain. To satisfy the causal connection requirement (b), we must show that if one of the sides of the connection changes the other side changes correspondingly. One direction follows immediately — changing the implementation of the contract object changes the behaviour of the system. Moreover, since the object represents a framework contract, the only legal way to change that part of the system behaviour is to change the corresponding contract. For if one changes the framework implementation by violating the framework contract, one forsakes the safety net captured in the framework design principles. So the other direction is satisfied as well, thus the causal connection requirement is satisfied.

The insight that an explicit representation of a framework contract is part of the meta-level, is vitally important as it leads to the conclusion that *the design of a framework provides an initial ground for the design of a meta-object protocol*. As a consequence we can be confident that the design of the meta-level interface can reuse parts of the design of the framework's base level interface. Knowing that the design of the meta-level is more difficult since iteration is more expensive, and knowing that there are lots of promising techniques for designing base level interfaces are under way, this is an extremely valuable perception in its own right.

Framework Design Guidelines

Knowing that explicit representations of framework contracts are meta-objects, we must say something about what contracts should be made explicit in the design. Obviously, we need some support to tell what contracts are pertinent, as a typical framework comes with numerous contracts. As argued below, the Zypher design space provides an ideal representation of the contracts that should be made explicit.

First, we propose a framework design methodology summarised in four design guidelines.

Framework Design Guidelines
1) Devise a design space for the problem domain.
2) Each design space axis should correspond to a black-box template method.
3) Each design space axis should correspond to a framework contract.
4) The configuration of the design space axes should correspond to a framework contract.

Second, we show that a system designer who respects these guidelines can build a system incorporating the three levels of tailorability.
1) The first guideline constrains our methodology to problem domains where it is possible to devise a design space. The Agora [Steyaert'94] and ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] experiments indicate that this is at least feasible.
2) The second design guideline ensures that a framework provides domain level tailorability. A software engineer who supplies a concrete implementation for the abstract methods called by the template method adds a point to the corresponding design space axis.
3) A framework that respects the third guideline, can be made system level tailorable by extending the design with meta-objects that are explicit representations of the framework contracts that follow from the guideline. By supplying a different contract object, one can provide extra behaviour for the execution of the contract, this way wrapping uniform behaviour around different points residing on the corresponding design space axis.
4) A framework that respects the fourth guideline, can be made configuration level tailorable by extending the design with a meta-object that is an explicit representation of the framework configuration contract. By supplying a different configuration object, one can change the configuration behaviour of the framework, this way changing the relationship between the axes of the design space.

Note that points 2, 3 and 4 imply the following implications

Object-Oriented Framework $\Rightarrow$ Domain Level Tailorability

Meta-Object Protocol $\Rightarrow$ System Level Tailorability

Meta-Object Protocol $\Rightarrow$ Configuration Level Tailorability

These implications provide the answer to the question how both object-oriented frameworks and meta-object protocols can be combined properly. In some sense, object-oriented frameworks and meta-object protocols are overlapping techniques since they both share the common goal of tailorability. Yet the techniques differ in the exact level of tailorability that can be achieved.

The final remark is about generalising the results for other kinds of open systems. Note that the domain dependent features are captured in the type and number of design space axes but that the framework design guidelines and the three levels of tailorability are completely independent of the type and number of design space axes. This leads to the conclusion that *the proposed framework design methodology can be applied on the construction of open systems in other domains than the hypermedia domain.* The constraining factor is the ability to construct a design space for the problem domain at hand.

Conclusion

We have provided an answer to the question how object-oriented software engineering can contribute to the development of open hypermedia systems. More precisely, we have argued how object-oriented frameworks and meta-object protocols can help to develop a hypermedia system with three levels of tailorability — domain level, system level and configuration level. By adopting those three levels of tailorability as distinctive characteristics of open hypermedia systems, we have concluded that object-oriented frameworks and meta-object protocols overlap, since they both share the common goal of tailorability, yet differ, since they achieve another level of tailorability.

The main point in our argumentation was the insight that *explicit representations of framework contracts are meta-objects*. Based on this insight, we have formulated four framework design guidelines and we have proposed a framework design methodology to incorporate the three levels of tailorability.

Moreover, we have argued that the same approach can be applied to open systems outside the hypermedia domain, because all our arguments are independent of the hypermedia domain. So if someone is able to devise a design space for a particular domain (that is, an n-dimensional axis system) and if that design space is turned into a framework that respects the framework design guidelines, then it is possible to incorporate the three levels of tailorability.

PART 2

The Zypher Design

CHAPTER **3**

The Zypher Software Artefact

In the domain of software engineering, it is common practice to validate techniques experimentally [NationalAcademy'94]. One builds a prototype for a particular application domain to investigate benefits and drawbacks of certain techniques and generalises the findings to a broader set of application domains afterwards. Such a prototype system is called a software artefact and in our dissertation this is role is played by the Zypher system.

Such an experimental validation methodology is summarised in the proverb "The proof of the pudding is in the eating". In the first part, we have been collecting a number of ingredients and instruments for a delicious pudding — the hypermedia design space, the three levels of tailorability, object-oriented frameworks and meta-object protocols. Also, we have been describing how such a delicious pudding might taste — the framework browser scenario. Finally, we have sketched a recipe to cook the pudding — the framework design methodology.

This second part, describes the cooking process we have carried out in order to cook a real pudding — the Zypher open hypermedia framework. This software artefact serves as a 'proof of concepts' that our framework design methodology is applicable for the construction of a concrete open system that incorporates the three levels of tailorability. Also, we describe the actual proof of the pudding, that is the eating process — the refinement of the Zypher framework into a prototype of a framework browser. This show how the three levels of tailorability helped us to tailor the open system for a prototype of a framework browser.

The Zypher Design Pattern Form

One of the main difficulties in describing a software artefact is that one must list a large number of issues and problems, relate this to the description of a software architecture and somehow convince the reader that this architecture provides a solution for the initial list of problems. This is a very difficult job that requires skills far beyond that of a typical software engineer.

The problem is even more difficult when describing object-oriented software architectures, because such an architecture must be reusable, thus address a number of similar, recurring problems. The object-oriented community acknowledges this problem and forwards design patterns as a way to describe reusable aspects of an object-oriented design. In a design pattern approach, instead of arguing how a particular architecture covers a complete list of problems, one argues how a small aspect of that particular architecture solves a subset of the initial list of problems. This way, certain aspects of that particular architecture can be reused in other architectures. Also, it is more easy to adapt the design to a changing list of problems. Using the design pattern approach, those reusable aspects are described in a systematic way to help authors and readers identifying the important facets; this is called the design pattern form.

There are several attempts to assemble design pattern classified in two main categories. Generic catalogues collect domain independent patterns; examples are [GammaEtAl'95] and [Pree'94]. Domain specific catalogues collect patterns about a specific problem domain; HotDraw [Johnson'92], [Beck,Johnson'94] is a well-known example for the domain of graphical editors.

In what follows, we describe the Zypher open hypermedia framework using the design pattern approach. So the second part of this dissertation is presented as a domain specific design catalogue for open hypermedia systems. We also include a description of the design pattern form as we have applied it in writing the Zypher design pattern catalogue.

The real power of design patterns lies in the idea of overlapping many of them on a small number of system elements. All the following patterns include numerous mutual references, to stress the way they work together. Nevertheless we want to stress that for a first tour, the sequential path is probably the best. Only for an in depth study, the referencing scheme is essential to understand how all patterns work together to achieve the overall design.

The Zypher design pattern catalogue is available in hypertext format as well; among others it has been used as part of the construction of a framework browser. Also, it is available on the world-wide web from the Zypher home page at *http://progwww.vub.ac.be/pools/Zypher/*.

Outlining the Zypher Design Pattern Form

All sections are mandatory, unless stated otherwise.

Title

The title names the pattern and provides a glimpse on the essence of the design. The title consists of a few keywords followed by a colon and a short phrase. The keywords constitute the name used in other design patterns as a reference and alludes to the problem or solution discussed in the design pattern. The short phrase condenses the core of the pattern and motivates its existence in the global problem.

Intent

The intent section summarises (2 to 4 lines) the design pattern. As such, the intent section elaborates on the short phrase in the title without mentioning the name. An icon reflects the level of tailorability (see the [puppet master metaphor]).

Analysis

The analysis section introduces the problem addressed by the design pattern and lists relevant issues. As such, the analysis section forms a bridge between the intent section and the reference in the title. The analysis section uses examples to illustrate how the problem is manifested in some problem domain without hinting at solutions. The keywords in the title are not necessarily mentioned in the analysis section; this depends whether the keywords refer to the problem or to the solution.

Problem

The problem section recapitulates the problem the design pattern tackles based on the issues addressed in the analysis section. The problem section is always stated as a question; usually a "How ?" question to emphasise that a solution is desired. The problem section is short and does not contain examples.

Solution

The solution section proposes a design that answers the question stated in the problem section. The solution section is written in a clear and concise style that emphasises on the proposed technique and/or structure. Normally, the examples taken from the analysis section are reproduced in the proposed design to demonstrate how it solves the problem.

Contract

The contract section includes charts (applying [class diagrams] or [object interaction diagrams]; see below) specifying the internal structure of the proposed solution. The rest of the contract section enumerates all elements of the charts (including relations between the elements) and charges them with precise responsibilities. Wherever possible, the messages that constitute the contracts are listed and the refinements subclasses are allowed to make are specified. The contract section does not contain any examples.

Motivation

The motivation section answers why the proposed solution solves the issues raised in the analysis domain. Among others, the motivation section includes alternative solutions and why they are rejected. The examples taken from the analysis and solution sections are reconstructed to illustrate certain advantages or drawbacks.

Issues (optional)

The issues section, if present, discusses issues subordinate to the ones listed in the analysis section. Typical issues handled in this section include naming, implementation remarks, relations with similar approaches, references to the literature, … .

Consequences (optional)

The consequences section, if present, discusses design trade-offs and includes references to other design patterns that handle the drawbacks raised by the introduction of the design pattern.

Relations

The relations section links the pattern with other design patterns in the text. This section also includes references to well-known design pattern catalogues. The section includes at least one 'default' reference, pointing to the most obvious next pattern to read. The reference is marked with a triangle symbol in the margin (∇).

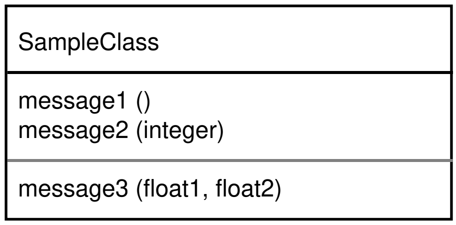
References

All the Zypher design patterns are organised in a mutual referencing structure. References are surrounded by brackets "[]" and use the name in the title (i.e. the keywords preceding the colon) to refer to another design pattern. To refer to a section within the pattern, the reference is followed by the ~ character and the title of the subsection.

Diagrams

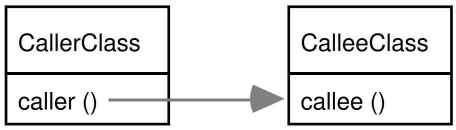
Throughout this document, class diagrams are represented using a variant of the "Object Modelling Technique" (OMT) notation. We refer to [Blaha,Premerlandi,Rumbaugh'88] and [RumbaugEtAl'91] for a full description of OMT. Below is a brief survey of the main elements in adapted notation.

Class & Messages



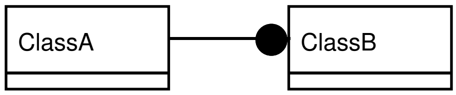
A class is denoted by a box with the class name at the top (i.e. SampleClass). A class name always starts with a capital. The set of messages understood by instances of that class appear below the class name; some messages may be separated by a dotted line to group related messages. Names of messages start with a small letter; names of parameters correspond with the name of the class but start with a small character (i.e. integer, float1, float2). In the example we see a parameterless message (i.e. `message1`), a one parameter message (`message2`) and a two parameter message belonging to another category (i.e. `message3`). Note that none of our classes have instance variables.

Calling Order



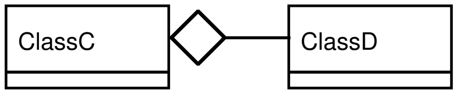
A grey arrow between to classes indicates a calling order from the CallerClass to the CalleeClass, meaning that an instance of CallerClass having received the message `caller` must send the message `callee` to an instance of CalleeClass. Note that, one of the participating messages may be left unspecified, which implies that this side of the calling order is not important in the design. The calling order is only shown when it is essential to the design being explained, as it quickly leads to cluttered schema's; Object Interaction Diagrams (see below) are preferred for larger calling structures.

One-to-Many Association



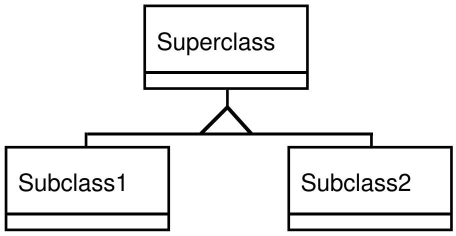
A line with a black circle indicates a one-to-many association from ClassA to ClassB, meaning that —at a certain moment— an instance of ClassA may be associated with multiple instances of ClassB. The line between the classes means that there is an association, the black circle means that this side of the association is the 'multiple' side. Usually, both classes understand messages to modify and query the association relation.

Aggregation



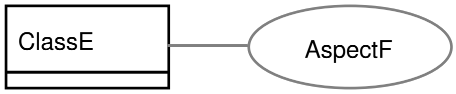
A line with a white diamond indicates an aggregation association from ClassC to ClassD, meaning that —at a certain moment— an instance of ClassC may contain multiple instances of ClassD. The line between the classes means that there is an association, the diamond means that this side of the associations is the 'container' side. Usually, only one class (the container class) understands messages to modify and query the aggregation relation.

Inheritance



A line with a white triangle indicates an inheritance relation with one superclass and two subclasses, meaning that all instances of Subclass1 and Subclass2 inherit the structure and behaviour of Superclass. Nevertheless, the structure and behaviour of the three classes may be different, since Subclass1 and Subclass2 may extend and override the structure and behaviour of the Superclass. The line between the classes means that there is an association; the top of the triangle points towards the superclass; the base points towards the subclasses.

Unspecified Aspect

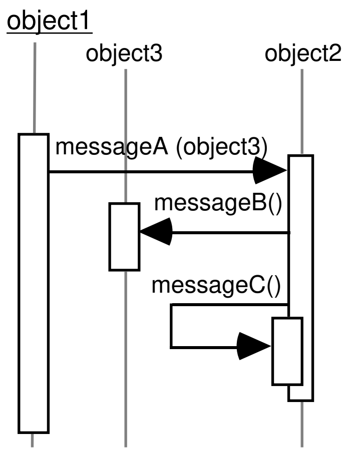


In some occasions, an aspect of the design is left unspecified. This means that its existence plays a crucial role in the overall design, but the actual structure is unimportant. The example shows that ClassE is related with an aspect AspectF, meaning that AspectF participates in the object configurations of ClassE, but that the actual structure of AspectF is yet unknown.

Interaction Diagrams

Throughout this document, object interaction diagrams are represented using an adapted version of the Objectory notation. We refer to [JacobsonEtAl'92] for a full description of the notation. Below is a brief survey of the main elements in the notation.

Message send



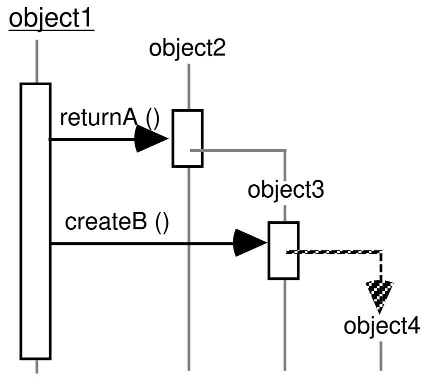
This is an example of an object interaction diagram illustrating message exchange between objects. The message involves three objects (object1, object2, object3) indicated by the vertical lines; the names of the objects are placed on top of the lines. One object (i.e. object1) is the originator of the interaction: its name is placed a little higher than the others and is displayed in a slightly larger and underlined font.

The time flows from the top to the bottom of the diagram. Vertical rectangles denote objects that are handle messages during a certain period: the bottom and top of the vertical rectangles denote the moment the object starts or ends handling the message. The messages themselves are depicted by horizontal arrows starting in the object sending the message and ending in the receiver of the message. The name of the message is written above the horizontal arrow; the names of the argument objects stand between braces.

The example shows an object1 sending a message `messageA` to object2 with object3 as parameter. While object2 is handling the message, it sends a parameterless `messageB` to object3 and thereafter it sends the `messageC` to itself. Since the bottom of the object1 rectangle is below the bottom of the object2 rectangle, control returns to object1 afterwards.

Return Object / Create Object

Existing objects returned as the result of a message send, are depicted with a line emerging from the message handling rectangle; the name of the object appears on top of the line (i.e. object 3). Objects created during a message send are depicted with an arrow emerging from the message handling rectangle; the name of the object appears just below the arrowhead (i.e. object4). To emphasise the difference, the object creation arrow is drawn in a dashed line while usual object lines are dotted grey lines.



The example shows an object1 sending a message `returnA` to object2 that returns an existing object3. Afterwards, object1 sends a `createB` message to object3 that creates an object4.

CHAPTER **4**

Data Structures For An Interoperable Hypermedia Framework

This part of the dissertation introduces the four central building blocks in the design of the Zypher Open Hypermedia Framework: component, anchor, instantiation and marker. The four building blocks are chosen as they support a modular architecture, easy to configure and easy to extend.

Those building blocks provides an initial design for the Zypher open hypermedia framework that is not related to the framework design methodology proposed in the Zypher contribution (p.63).

The Dexter Model

Readers familiar with hypermedia research recognise the names of the four building blocks as the main elements of the Dexter Hypertext Reference Model ([Halasz,Schwartz'90]). One of the original goals of the Dexter Hypertext Reference model was to define a hypermedia interchange standard for a wide range of existing and future hypermedia systems. The model attempted to capture the state of the art of that time's most prominent hypermedia systems in a written specification. As such, the Dexter model had a considerable impact on the design of hypermedia systems; some hypermedia systems even initiated their design using the Dexter specification (notably DeVise Hypermedia [Grønbaek,Trigg94] and the Amsterdam Hypermedia Model [Hardman,Bulterman,VanRossum94]).

The design of the Zypher Open Hypermedia Framework builds on the accumulated knowledge of the Dexter specification as well. This may seem awkward at first, as the Dexter model is generally regarded as a representative of the generation of monolithic hypermedia systems (see [Wiil,Østerbye'94], [Legget,Schnase94]), whereas Zypher explicitly targets the domain of Open Hypermedia Systems. The design patterns in this chapter, illustrate that the four basic building blocks deal with many of the issues involved in open hypermedia systems, among others because they adhere to the basic design principles forwarded in the flag taxonomy of open hypermedia systems (see p.27 and [Østerbye,Wiil'96]). Other issues are dealt with in subsequent chapters.

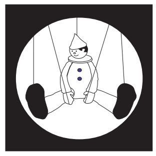
Structural Objects

Data structures are the central theme underneath the design patterns in this part. All the design patterns listed here, describe so-called structural objects; elements for Zypher data structures. When assembled properly, the Zypher data structures are internal representations for external information. To emphasise this representation relationship, the examples in this part emphasise on how framework information can be represented with the structural objects. The behaviour of the structural objects is beyond the scope of the design patterns in this part.

Each of the four structural objects represents a wide variety of possible implementations, captured in the inheritance hierarchy they represent. Each structural object defines a public interface to be used by clients; inheritance guarantees that implementations are compliant with the public interface.

Interoperability: Unaware Repositories and Applications

Intent



Define a data model to manage information structures stored in various repositories and manipulated using different applications, all unaware of each other.

Analysis

Information managed with computers is heterogeneous in nature: it resides in various *information repositories* and may be manipulated using different *viewer applications*. What those repositories and viewer applications are, depends on the domain the information describes.

Besides heterogeneity, there is also the flexibility requirement. Additional needs demand for extra repositories and viewer applications; new technology and better software emerge and must be integrated.

EXAMPLE: Integration of analysis, design and implementation tools

> Object-oriented frameworks are complex information structures, mixing an analysis, design and implementation for a particular problem domain. A *framework browser* combines current and future analysis, design and implementation tools to attain an overall functionality that is larger than the sum of its parts.
>
> Framework analysis and design information resides in design pattern documentation stored in files on some file system (possibly a file server shared by all the participants of the framework development team) and sometimes in design pattern catalogues published on the world-wide web. Design information is also available from the abstract classes and protocols of the framework, while implementation information can be found in the concrete classes of the framework and in cookbook documentation. Thus the information repositories for frameworks are the file system, the world-wide web and the framework development environment. Note that the latter must provide functionality to maintain design pattern relations.
>
> Different tools manipulate the analysis, design and implementation information. There exist implementation oriented tools like code browsers, version management tools, debuggers, code analysers and cookbooks; there exist analysis and design oriented tools like design diagramming tools, design pattern catalogues and contracts. As frameworks are designed for a specific problem domain, one might expect that domain specific tools exist as well (i.e. a framework for multi-media incorporates multi-media authoring tools, a framework for decision support systems incorporates mathematical packages, …). New tools (i.e. pattern browsers, contract analysers, …) emerge as the expertise of building frameworks grows and such tools should be able to be integrated in the framework as well.

The Zypher Framework Browser is designed for the VisualWorks\Smalltalk environment and incorporates implementation oriented tools (i.e. the standard Smalltalk code browsers, some special Zypher code browsers), design tools (design pattern browsers) and design and analysis tools (design pattern documentation manipulated using a text editor, a HTML-editor and the Microsoft Word third party application).

A common aspect of information structures is complexity. To master complexity, people normally use abstractions that show important ideas and hide details. But ideas considered details may become woefully important from another perspective. To cope with this problem, information systems provide context dependent abstraction views for an information structure.

EXAMPLE: Code browsers and pattern browsers

A framework is a complex information structure that needs various context dependent abstraction views. A framework consists of a set of co-operating objects grouped in classes defining messages and methods. From this definition follow several sensible abstraction views, all grouped under the name [code browser]. One abstraction view consists of all the classes that make up a framework (this is called the [system browser]), another one consists of all the messages and methods in a class (typically called the [class browser]), a third one consists of all the methods that implement a particular message (the [implementors browser]), and another one consists of all the methods that send a particular message (the [senders browser]). These are all implementation oriented abstractions and all Smalltalk development environments provide such abstraction views. The Zypher framework browser incorporates all code browsers that come with the standard Smalltalk environment.

A framework is a reusable design, specified using design patterns. Design patterns are captured partially in design pattern documentation and partially in contracts between abstract classes of the framework. Abstraction views for design patterns should be able to incorporate both parts. As far as the design pattern documentation is concerned, the quality of the abstraction view depends largely upon the skills of the author. It is the author that must set-up the information perspectives when writing the documentation, although the design pattern form is believed to be helpful in this process.

The abstractions provided in the source code of an object-oriented framework do not aim at different perspectives: they aim at a modular organisation of the software to attain flexibility and reusability. This is reflected in the tools that come with the development environment: code browsers provide a narrow, fine-grained view on the framework. This conflicts with design patterns, which are usually spread over several classes and include but a few methods per class. So to inspect the design pattern contracts between abstract classes we need specially designed browsers. That is why Zypher provides different implementation perspectives by means of the so-called pattern browsers. A pattern browser can be used to inspect and modify the protocol methods of different classes at the same time in the same browser.

The heterogeneity and flexibility requirements imply that the set of repositories and viewer applications that form an information system are not known in advance. This raises the issue of *unawareness*: the subsystems of an information system do not know about the other participants in the overall system. This is to say that information repositories are not aware of other information repositories, nor the viewer applications that manipulate the information they provide. Viewer applications do not know about other viewer applications but are aware of the information repositories that contain the information they manipulate.

EXAMPLE: Unawareness of repositories and viewer applications

A framework browser involves information repositories like the local file system, the world-wide web and the framework development environment, that do not contain direct references towards each other. Moreover, the repositories do not know what viewer applications (i.e. the standard Smalltalk code browsers, some special Zypher code browsers, design pattern browsers and various text processing tools) manipulate the information they provide. Also, the viewer applications do not know what other viewer applications are contained in the framework browser.

The problem of unawareness (i.e. software that is not explicitly designed to co-operate and yet must do so), is generally referred to as the *interoperability* problem. A system that deals with the problem is called an *interoperable system*.

Problem

How can one model an interoperable information system where several viewer applications can manipulate information residing in different repositories ? How can this model respect the unawareness constraint ?

Solution

An interoperable information system manipulates information fragments where each information fragment incorporates two important aspects. The storage aspect concerns the internal structure of the information fragment and the way it is addressed in the repository. The presentation aspect concerns attributes specifying how an information fragment should be presented and manipulated by a viewer application.

To model an interoperable information system, separate these two aspects of an information fragment in two classes of objects. These objects mediate between the viewer application and the repository.

Call the object that models the storage aspect a *component*; a component communicates with the repository to map repository addresses on the corresponding contents. Call the object that models the presentation aspect an *instantiation*; an instantiation commands the viewer application to present the contained information.

Translate the *unawareness constraint* in contracts stating that components are not allowed to send messages to other components or instantiations directly, and that instantiations are not allowed to send messages to other instantiations directly.

EXAMPLE

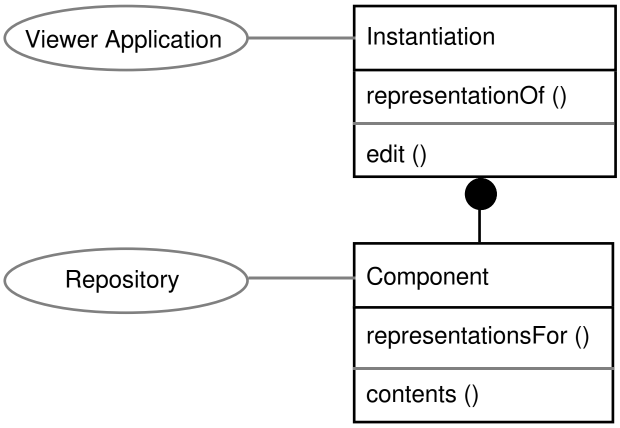
The Zypher framework browser incorporates among others the following viewer applications: a text editor, the Microsoft Word third party application, a HTML-browser, the standard Smalltalk code browsers (system, class, senders and implementors), three Zypher code browsers (class, senders and implementors) and numerous pattern browsers.

The information repositories are the world-wide web (HTTP servers), the local file system, the Smalltalk repository (containing all the classes and methods known in the Smalltalk system) and the pattern fact base (maintaining what methods and classes participate in which design pattern).

The presentation aspects can be modelled with nine classes of instantiation objects. A Microsoft Word instantiation represents a Microsoft Word document; a HTML-instantiation represents a HTML-document; a text instantiation represents a text document and there is one instantiation for every standard Smalltalk code browser (system, class, senders and implementors). To model the Zypher code browsers and the pattern browsers, a different approach is chosen. Instead of modelling all browsers as separate instantiations (this would be hard since the number of pattern browsers is quite large and framework dependent) model the panes of the browser. Zypher provides a source code pane and a list pane instantiation. Those panes can be assembled in various combinations to form the desired code and pattern browsers.

The structural aspects can be modelled with five classes of component objects. A file component maps file names (i.e. addresses in the file system repository) on streams, which may contain a Microsoft Word, text or HTML-document). A HTTP component maps an HTTP address (i.e. an address in the WWW repository) on a HTML-stream (containing a text or HTML-document). A Smalltalk class component maps a class name on the corresponding Smalltalk class; a Smalltalk method component maps a class name and method selector on a Smalltalk method. A pattern query component maps a query and a set of parameters on a list of class names and method selectors.

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Viewer Application

The actual contract concerning the viewer application is handled in the [editor] pattern.

Repository

The actual contract concerning the repository is handled in the [loader] pattern.

Component

A component models the storage aspect of an information fragment. All components implement the `contents` method that communicates with the repository to return the actual information (see [loader ~ contracts] for further details). Specific component classes may extend the protocol with messages that specify the address of a particular information fragment and the desired return format. Component objects are forbidden to send messages to other components or to their associated instantiations directly (see [events ~ contract] to learn how they synchronise).

Instantiation

An instantiation models the presentation aspect of an information fragment. All instantiations implement the `edit` method that commands the viewer application to present the instantiation (see [editor ~ contracts] for further details). The information being presented should be based on the contents of the associated component, as returned by the `contents` message. Specific instantiation classes may extend the protocol with messages that specify the attributes controlling the presentation and the behaviour of a particular information fragment. Instantiation objects are allowed to send messages to their associated components, but not to other instantiations (see [navigation] to learn how they interact).

Component / Instantiation Association

There exists a one-to-many association between a component and an instantiation. This association should be interpreted that, at a certain moment, a single component may be represented by several instantiations but that a single instantiation is allowed to represent only one component. However, the association is allowed to vary over time. All components implement the `representationsFor` method to enumerate all the instantiations associated with the component. All instantiations implement the `representationOf` method to answer the component associated with the instantiation. See [meta-objects ~ contracts] for further details.

Motivation

Since the viewer application and repository are in principle unaware of each other, there should at least be one object that mediates between the two. This object should translate some viewer operations (i.e. save) in calls to the repository and some repository operations (i.e. change) in calls to the viewer application. Then why have two mediator objects instead of one ?

a) Configurability

The most important argument to separate storage and presentation is for easier configurability. A single piece of information may be available from different repositories and may be manipulated using different viewer applications. According to the system's configuration, one combination may be preferred over another. It is cheaper to assemble a combination at run-time than it is to construct all possible combinations in advance and choose one of them. Also, if such run-time options are separated in different classes it is easier to configure the system because one option affects one class only. Finally, if the storage aspect is separated from the presentation aspect, it is possible to change the combination at run-time.

EXAMPLE

The Zypher framework documentation is organised as a collection of design patterns. Each design pattern comes in a HTML and a Microsoft Word version. The Microsoft Word version is richer (i.e. can be edited, contains pictures) but is not available on UNIX platforms. On such platforms, software engineers must use the HTML version. Note that certain preference settings use the HTML version by default.

When preferring HTML as the documentation format, there are again two options. For some design patterns, software engineers prefer to store the HTML-files on their local file system for faster access while other design patterns may be centralised on a shared HTTP server to keep them up to date with the rest of the project group. Moreover, if the system monitors that a particular HTTP address is consulted often and changes rarely, the corresponding HTML-stream might be transferred to local file.

b) Extensibility

The second closely related motivation is extensibility. When the mediators are separated, it is possible to reduce the number of classes and to assist extensibility by promoting the reuse of implementation. To explain this phenomenon, note that a single viewer application manipulates information residing in different repositories and a single repository may serve different viewer applications. If the storage aspect is separated from the presentation aspect then it is possible to reduce the number of mediator classes by factoring out the behaviour along the two dimensions. The worst case scenario for having one mediator object is that one needs to create m * n classes (where m is the number of repositories and n is the number of viewer applications) while one can do with m + n classes if the mediator is separated in two classes. Of course, if few information repositories can co-operate with few viewer applications then the number of classes is not reduced. But even then —for modest extensions to the system— it is easier to reuse the implementation by creating subclasses, since the implementation of one class focuses on a single aspect.

EXAMPLE

The Zypher framework browser incorporates the Microsoft Word application as a text editor and the HTML-browser as viewer applications for framework documentation. All manipulate information residing in files but the HTML-browser and text editor may incorporate information coming from a HTTP server as well. If we capture the mediation in a single class then we would have five mediator classes (Microsoft Word/File; Text/File; HTML/File; Text/HTTP; HTML/HTTP), separating the mediator needs five as well (a Microsoft Word instantiation, a HTML-instantiation, a text instantiation, a file component and a HTTP component). If we extend the Zypher framework browser to incorporate an FTP server as information repository, then the creation of one extra class is be sufficient to allow all the viewer applications to manipulate information residing on a FTP server (see the greyed column in [figure 18]). Moreover, the FTP component class would have the same interface and a similar implementation as the file component, so it probably would become a subclass of the file component class. In fact, all information repositories that are able to cache their information in the local file system may benefit in the same was as the FTP server does: they subclass the file component and override some key methods to incorporate the caching behaviour.

A similar example can be found in the pattern browsers and the Zypher code browsers. All list panes (i.e. the list of all the methods belonging to a class; the list of all the messages sent by a method; the list of all the implementors of a particular message; the list of all the classes or methods that play a given role in a particular design pattern) are manipulated using the same list instantiation class but work on different component classes. The source code panes are manipulated using the source code instantiation and work on the class and method component. Again the number of classes is equal in both cases, but if we extend the Zypher framework browser to incorporate a formatted source code viewer (one that automatically indents the source code for better readability) then adding a single class would do (see the greyed row in [figure 18]). Moreover, such an adapted source code viewer may benefit from subclassing the standard source code viewer, as the class has the same interface and a similar implementation as the standard source code viewer.

Component \ Instantiation	File	HTTP	FTP	Class	Method	Pattern Fact
MS Word	x		x			
HTML	x	x	x			
Text	x	x	x			
List				x	x	x
Source code				x	x	
Formatted code				x	x	

Figure 18: Combinations of components and instantiations

c) Dynamics

The final argument to separate the structure and the behaviour in two different classes is to cope with *dynamics*. The same piece of information may have different co-existing representations in different viewers. Certainly in an environment where information is shared between co-operating people it is likely that several persons manipulate the same information on the same moment. If one viewer modifies a piece of information then all other viewers manipulating this piece of information must be notified so they are able to update their display. This behaviour is easy to model with a one-to-many association between a component an instantiation: one defines a contract stating that when a component is modified all associated instantiations are notified as well. This is similar to the observer pattern described in [GammaEtAl95].

EXAMPLE

> The same HTTP component may be opened simultaneously by a HTML-instantiation and a text editor. If the text editor modifies the contents of the document, the HTML-browser should update the display.
>
> Almost all code and pattern browsers maintain a list of the messages sent by a particular method manipulated in a source code pane. If the source code pane modifies the source of a method then the list of messages sent by the method needs to be updated.

Issues

Naming

Names like component and instantiation may look unfamiliar to people outside the hypermedia community. They stem from the Dexter model [Halasz,Schwartz'90], a reference model that had a considerable impact on the design of hypermedia systems.

Design

When is something an instantiation ? When is something a component ? This is part of the design of the information system and is influenced by the available information repositories and viewer applications, and by the amount of flexibility required by the information system designer.

EXAMPLE

> It was an explicit design decision to model all panes of pattern browsers and Zypher code browsers as separate information fragments. This makes it easy to assemble new pattern browsers, which is necessary in the realm of framework browsers since design patterns are to a large extent framework dependent and so must be the pattern browsers. On the other hand, the subviews of the standard Smalltalk code browsers are not represented as objects in the Zypher framework, since this would require a full rewrite of all the code implementing the standard Smalltalk browser.

Consequences

An implication of modelling one information fragment with two objects representing the storage and presentation aspect, is that both aspects must be kept in sync. Since the unawareness constraint states that a component (storage aspect) is not allowed to communicate directly with an associated instantiation (presentation aspect), synchronisation messages are prohibited. This problem is tackled in the [events] design pattern.

As shown in the [motivation] section, not all components are supposed to work with all instantiations. When assembling an internal representation for external information, the hypermedia system must decide which component or instantiation to create. This issue is dealt with in the [meta-meta-objects] design pattern.

The extensibility argument in the [motivation] section claims that separating the aspects avoids the proliferation of classes, i.e. $m + n < m * n$. However, for small numbers, $m + n$ is almost equal to $m * n$. In such cases, the effect of separating the aspects is small and results in a more difficult design. There is no way to circumvent this problem.

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned how the introduction of a component and instantiation object solves the interoperability problem. The obvious next step is then the [navigation] pattern, which explains the fundamental operation in all hypermedia systems.

Some people may wonder about the unspecified aspects in the contracts section (i.e. repository and viewer application). These are explained in the [loader] and [editor] patterns.

The [events] pattern explains how instantiations synchronise with their associated components without violating the unawareness constraint (i.e. components are not allowed to send messages to their associated instantiations directly).

The one-to-many association between a component and an association is managed by the session meta-object explained in the [meta-objects] pattern.

The [meta-meta-objects] pattern provides help to set-up a configuration of components and instantiations.

Components and instantiations play an important role in the navigation template as described in the [navigation template].

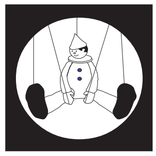
Other Catalogues

The messages defined in the contracts section participate in the template method pattern as defined in [GammaEtAl'93] and [Pree'94]. To use the terminology of the latter, `contents` is a hook method for the `edit` template method and together they form the 1:1 connection meta-pattern. The `edit` message (as well as the `representationOf` message) are hook methods for the navigation template [navigation template].

The component plays the subject and the instantiation plays the role of the observer in the observer pattern defined in [GammaEtAl'93].

Navigation: Provide Seamless Integration

Intent



Define a data model that assists in the exploration of complex and dynamic information structures by offering seamless integration between different applications.

Analysis

One important difficulty with interoperable information systems (see [interoperability]) is the integration of the different applications that constitute the system. Such applications may offer different perspectives on complex information structures, but that alone is not enough to really assist in the management of complex information structures. People must spend too much mental effort to instruct the different applications (each may come with a different command set) to show the desired perspective; too little of their mental capability is left for the actual task of managing the complex information structure.

EXAMPLE

An object-oriented framework is a complex artefact, where a single element (i.e. a class or a method) captures many facets of the target problem domain. Design patterns provide the necessary abstractions that combine the analysis, design and implementation of a particular aspect of the problem domain. To understand the design patterns underlying the framework, a software engineer consults all the available information sources, which implies that at least two viewer applications are involved (one code browser and one documentation viewer). Each viewer application has a different way to express what information fragment is to be shown (i.e. in a code browser this is done by selecting class- and method names from pre-compiled lists, in a documentation viewer one must map a reference to a design pattern on a location within a file). As a result, a software engineer reading design pattern documentation that refers to a particular method in a particular class must open a code browser, select the class in the class list pane and select the method selector in the method list pane to see the desired code appear in a source code pane. Each of these steps involves several searches and various choices which all require the software engineer to use part of his mental capacity. The reverse direction is even worse, since general purpose documentation tools like Microsoft Word do not support the notion of design patterns: it is the software engineer who maps a reference to a particular location within a particular file. Such a mapping operation requires even more mental effort, since it involves the interpretation of mnemonic file and section names.

Even if one remains within the boundaries of the framework development system the seamless integration between different abstraction views remains a problem. In framework research it is generally agreed that the best designs have many design patterns that overlap on the same elements. To exploit the full power of the framework it is necessary to understand the different design patterns each element participates in,

> but for a given framework element a pattern browser offers only a single design pattern perspective (i.e. one role in one design pattern). To see other design pattern perspectives, one must open another pattern browser on the given element. Here again several searches and choices are involved: what are the other design patterns an element participates in, how to open the according browser, what is the role this element plays in the pattern, how to instruct the pattern browser to display that role ? The general purpose design pattern catalogue [GammaEtAl95] already includes twenty-three design patterns and a particular framework is supposed to add other special purpose ones, so one might expect about thirty possible pattern browsers, each with about four different roles. Again, a software engineer has to perform to many complex actions to open the desired browser on the desired element.

This explains an extra requirement for an information system managing many abstraction perspectives on complex information structures: *seamless integration*. Users must be able to shift swiftly from one perspective to another, without being burdened with numerous choices and searches.

The hypermedia research domain tackles the seamless integration problem by means of *navigation*. Users select an item of interest and issue some kind of "follow link" command that is translated into an information request for the underlying hypermedia system that opens new perspectives on related items. The hypermedia system takes care of the search for appropriate information and the opening and instruction of the application to display this information. With hypermedia navigation, users are not confronted with difficult searching activities and do not need to perform lengthy command sequences to open a new information perspective.

EXAMPLE

> The Zypher framework browser provides different information perspectives with several viewer applications (i.e. code browsers, design pattern browsers, documentation browsers; see [interoperability: combining unaware repositories and applications]). It is possible to start navigation actions from all viewer applications. In the standard Smalltalk code browser one may issue a 'senders' or 'implementors' request to open the corresponding Smalltalk code browser on the selected method. The Zypher code browsers have similar functionality, but offer extra facilities for opening design pattern browsers or design pattern documentation that describe the item being browsed.
>
> Pattern browsers can be used to explore the key methods of different classes that participate in a certain design pattern. Pattern browsers contain list panes for each role assigned to a design pattern participant. The items in the list are the elements of the framework that may perform the role. Software engineers select the items of interest to see the actual implementation displayed in the source code view associated with the list pane. Also, each pattern browser comes with a context sensitive pull down menu that brings a software engineer to other design pattern browsers on the selected item or to design pattern documentation describing the role of the element in the pattern. From the documentation browsers (i.e. HTML and Microsoft Word) it is possible to navigate to other parts of the documentation, code browsers or design pattern browsers.

The hypermedia research has adopted its own terminology to talk about navigation. A navigation action starts in a certain location called the *source*, and ends in another location called the *target*. Each application is supposed to *highlight* potential sources and *select* targets to provide user feedback. The connection between the two endpoints is called a *link*; navigation is sometimes referred to as the traversal of a link. All links are maintained by the *link engine* of the hypermedia system.

EXAMPLE

With the HTML-browser, sources for navigation actions are highlighted by means of underlining. A simple mouse click on an underlined part is all that is needed to start a navigation action. Targets of navigation actions are selections within a document.

The Microsoft Word case is a little bit more complex. Sources for navigation actions are highlighted using underlining and colouring (blue). To start a navigation action one must select some underlined coloured part of a Microsoft Word document and choose the 'follow Zypher link' in the edit menu. Targets for navigation actions are ordinary selections provided by the Microsoft Word application.

In the Smalltalk and Zypher code browsers the sources and targets are methods selected in the method list pane. To start a navigation action one must choose the 'senders' or 'implementors' item from the pop-up menu associated with that method list pane. The Zypher code browser also allows to open a pattern browser on the selected method by choosing the name of the design pattern and role from a context sensitive 'design patterns' menu.

The navigation sources and targets for pattern browsers are selections in the list and source code panes of the Pattern browser. Pattern browsers implement some kind of local navigation, in the sense that changing the selection in one list pane changes the contents of other panes and thus the perspective offered by the pattern browser itself. Pattern browsers also offer global navigation by context sensitive menus that bring software engineers to other design pattern browsers or to design pattern documentation.

Note that the unawareness issue must be considered here as well. Since viewer applications are unaware of the other viewer applications, the link engine must be considered as a stand-alone entity and viewer applications are not allowed to call the services of the link engine directly.

Problem

How can one incorporate the notion of navigation into the model of an interoperable information system (see [interoperability]) to provide seamless integration between the various information views ?

Solution

To model sources and targets of navigation, one must be able to refer to substructures of information fragments. So from the navigation point of view, an information fragment is an aggregation of substructures. Just as the information fragment presented in the [interoperability] pattern, substructures have a structural and a presentation aspect. These aspects are separated in two kinds of objects.

Call the object that models the structural aspect an *anchor*. An anchor mediates between the repository and the link engine and communicates with the component and the repository to map a reference (an index, a name, a range) on a value. This value must be interpreted by the link engine to return the target of navigation operations. Call the object that models the presentation aspect a *marker*; it is the responsibility of the marker to mediate between the viewer application and the link engine. A marker is able to command the viewer application to highlight subparts of instantiations and is activated by the viewer application to start a navigation action.

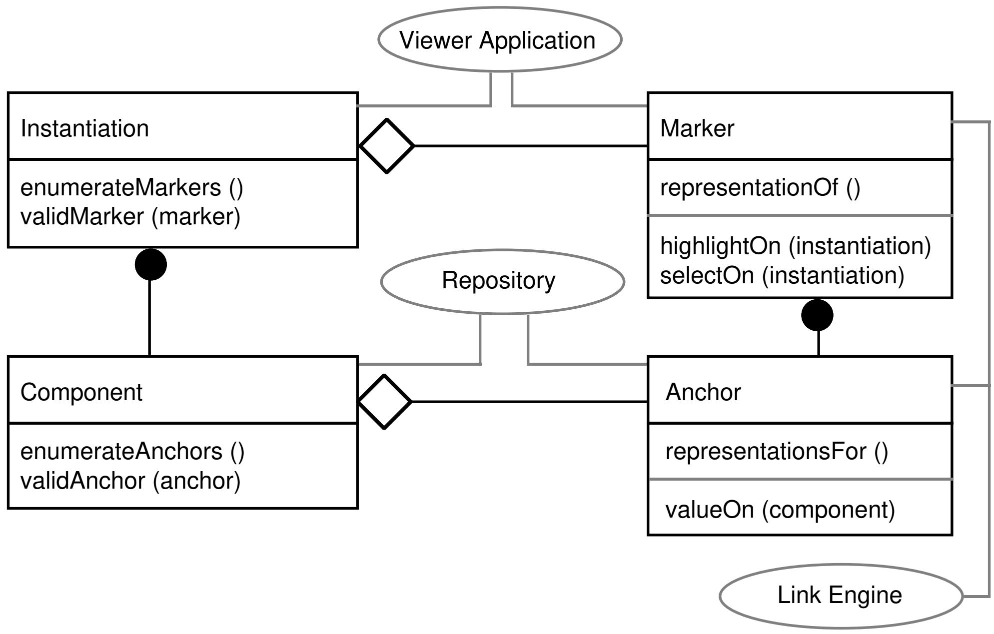
Translate the *unawareness constraint* in contracts stating that components / instantiations are not allowed to send messages to anchors / markers directly; that anchors are not allowed to send messages to markers or other anchors directly and that markers are not allowed to send messages to other markers directly. In this way, the marker - anchor pair mediates between the viewer application and the link engine.

EXAMPLE

To model the navigation actions described in earlier examples, the Zypher framework browser incorporates three types of anchors. An index anchor refers to an item in a list of items and is used for the lists inside the pattern and code browsers. A range anchor refers to an (optionally discontinuous) range of characters in a text and is used for source anchors in the HTML-browser and Smalltalk source code panes[9]. A value anchor refers to an anchor with a value that is maintained by the viewer applications (i.e. bookmarks in Microsoft Word and target anchors in HTML).

There are also several markers involved: a list marker is the source or target for the selection of an item from a menu or in a list pane. A sensitive text marker identifies portions of text in the HTML-browser or in a Smalltalk source pane and the active bookmark marker is used with the Microsoft Word application.

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Link Engine

The actual contract concerning the link engine is handled in the [resolver] pattern.

Component

Component objects are not allowed to send messages to their associated anchors.

Anchor

An anchor models a substructure of a component. All anchors implement the `valueOn` method that returns a value to be interpreted by the link engine. Specific anchor classes may extend the protocol with messages that specify the format of the reference to the substructure. Anchor objects are allowed to send messages to the components they are contained in; these container components are always passed as parameter. Anchor objects are forbidden to send messages to their associated markers. See [loader ~ contracts] for further details.

9 Note that Smalltalk uses a syntax where messages are written with keywords separating arguments. So a single message selector is spread over a discontinuous range of text.

Instantiation

Instantiation objects are not allowed to send messages to their associated markers.

Marker

A marker models an endpoint (source and/or target) of a navigation operation. All markers implement the `highlightOn` message that instructs the viewer application to highlight the given substructure as a candidate source for a navigation operation. All markers implement the `selectOn` message that instructs the viewer application to select the marker as the target of a navigation operation. Specific marker classes may extend the protocol with messages that specify the attributes controlling the presentation and the behaviour of particular sources and targets. Marker objects are allowed to send messages to the instantiations they are contained in; these container instantiations are always passed as parameter. Marker objects are allowed to send messages to their associated anchors. See [editor ~ contracts] for further details.

Anchor / Marker Association

There exists a one-to-many association between an anchor and a marker, equivalent to the one-to-many association between a component and an instantiation. This association should be interpreted that, at a certain moment, a single anchor may be represented by several markers but that a single marker is allowed to represent only one anchor. However, the association is allowed to vary over time. All anchors implement the `representationsFor` method to enumerate all the markers associated with the anchor. All markers implement the `representationOf` method to answer the anchor associated with the instantiation. See [meta-objects ~ contracts] for further details.

Component / Anchor aggregation

From the viewpoint of navigation, a component models an aggregation of anchors. This aggregation should be interpreted that, at a certain moment, a component contains a number (zero or more) of anchors and that a certain anchor may be contained in several components. The aggregation is allowed to vary over time.

All components implement the `enumerateAnchors` method to enumerate all the anchors in the aggregation and the `validAnchor` method to check whether an anchor is part of the aggregation. See [meta-objects ~ contracts] for further details. It is not possible to ask an anchor what components act as containers: for all anchor operations that matter, the component is passed as an argument.

Instantiation / Marker aggregation

From the viewpoint of navigation, an instantiation models an aggregation of markers. This aggregation should be interpreted that, at a certain moment, an instantiation contains a number (zero or more) of markers and that a certain marker may be contained in several instantiations. The aggregation is allowed to vary over time.

Moreover, the aggregation relation mirrors the Component/Anchor aggregation relation. A marker M that is contained in an instantiation I should be associated with an anchor A that is contained in a component C, associated with the same instantiation I.

All instantiations implement the `enumerateMarkers` method to enumerate all the markers in the aggregation and the `validMarker` method to check whether a marker is part of the aggregation. See [meta-objects ~ contracts] for further details. It is not possible to ask a marker what instantiations act as containers: for all marker operations that matter, the instantiation is passed as an argument.

Motivation

Navigation forces us to make the aggregation relationship between an information fragment and the source or target of a navigation action explicit. This explains why there should at least exist one object that represents a referable substructure of an information fragment. Of course this separation makes it possible to reuse the same referencing technique with different kinds of information fragments.

EXAMPLE

> The most obvious technique to refer to a substructure of a text is to maintain the positions of the first and last character of a range of text. This is exactly what the range anchor does and as such it can be used on both HTML components and Source code components. To cope with discontinuous ranges, the range anchor contains a list of ranges.
>
> An element of an ordered collection can be referred to by its index. This explains the existence of the index anchor working on a list component. The list component can be presented using a list instantiation or a menu instantiation, but all work with the same list marker.

The question is then why to model a single substructure with two separate objects ? The same arguments that made us separate the storage and presentation behaviour of an information fragment (see [interoperability]) apply (i.e. configurability, extensibility and dynamics).

Issues

Naming

Just like in the [interoperability] pattern, the terms anchor and marker stem from the Dexter model [Halasz,Schwartz'90].

Design

When should an operation be considered as a navigation action ? That is part of the design of the hypermedia system and is left as an option. As with all options, a trade-off must be considered. Defining a certain operation as a navigation action involves more work for the software engineer and requires more system resources. On the other hand, defining operations as navigation actions has certain benefits for the users of the hypermedia system (i.e. maintaining a trail of visited locations; see [meta-objects]).

EXAMPLE

> Opening a design pattern document (with the Microsoft Word application or with the HTML-browser) is not normally considered a navigation action. However, defining it as a navigation action enables the hypermedia system to cache the contents of the document to improve speed.
>
> If the request for the senders or implementors of a message inside a code browser is regarded as a navigation action, then the hypermedia system may log this activity to provide back facilities.

Consequences

The synchronisation issue, described in [interoperability ~ consequences], is even more relevant in the adapted model, since there are more parties involved. The issue is handled in the [events] design pattern. The same goes for the configuration issue, handled in the [meta-meta-objects] design pattern.

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned how the introduction of an anchor and marker object allows to incorporate navigation. The obvious next step is then the [resolver] pattern, which explains the elements of an extensible link engine.

The [events] pattern explains how markers, anchors, instantiations and components synchronise their contents without violating the unawareness constraint.

The one-to-many association between an anchor and an instantiation is managed by the session meta-object; the one-to-many association between a component / instantiation and an anchor / marker is managed by the path meta-object. These are explained in the [meta-objects] pattern.

The [meta-meta-objects] pattern provides help to set-up a configuration of components, anchors, instantiations and markers.

Anchor and markers play an important role in the navigation template as described in the [navigation template].

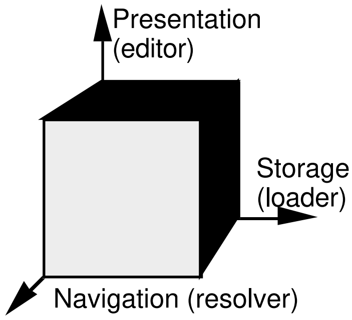
Other Catalogues

The messages defined in the contracts section participate in the template method pattern as defined in [GammaEtAl'93] and [Pree'94]. To use the terminology of the latter, `valueOn` is a hook method for the `highlightOn` and `selectOn` template methods and together they form the 1:1 connection meta-pattern. The `highlightOn` and `selectOn` messages (as well as the `representationOf` message) are hook methods for the navigation template [navigation template].

The component / anchor, instantiation / marker and anchor / marker form observer patterns as defined in [GammaEtAl'93].

CHAPTER **5**

A Design Space For A Hypermedia Framework



This chapter introduces three behavioural objects in the design of the Zypher open hypermedia framework: resolver, editor, and loader.

The introduction of the behavioural objects is the first step towards modelling hypermedia systems using the Zypher design space (p.30). This design space is delineated by three orthogonal design axes, where each axis represents aspects important for all hypermedia systems, namely navigation, presentation and storage. A hypermedia system is then a three dimensional volume in that design space.

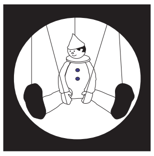
Based on the second of our framework design guidelines, stating that "Each design space axis should correspond to a black-box template method" we introduce a behavioural object to represent a point on one of these design axes (i.e. resolver for the navigation axis, editor for the presentation axis and loader for the storage axis). This allows us to incorporate the notion of domain level tailorability into the design of the framework.

Behavioural Objects

Behaviour is the central theme underneath the design patterns in this chapter. All the design patterns listed here, describe objects representing behaviour for Zypher data structures. Behavioural objects are an internal representation for (possible third party) software that is supposed to store, present or navigate the information structures.

The design patterns in this chapter are quite similar. This should not come as a surprise as the same basic technique is at work here: to model a certain design space axis one favours the black-box approach over the white-box approach.

Resolver: The Core of an Extensible Link Engine



Intent

Encapsulate the algorithms that determine the navigation relations (i.e. what targets can be reached from a given source) to accomplish a hypermedia system with an extensible link engine.

Analysis

Navigation is an answer to the problem of seamless integration, but the solution introduces a new problem: how to manage the connections between related information ? Traditional hypermedia systems use static links for these purposes, which means that the author of a hypermedia system must maintain the connections between sources and targets of navigation relations manually. Static link creation is usually accomplished by some kind of recording mechanism, where users select parts of information fragments in viewer applications and issue a command to mark the selection as the source or target of a link.

Static links have the advantage of being applicable in all kinds of information structures, since they are independent of the problem domain: it is the author of a link that determines the relations between the information fragments and provides the semantics. However, maintenance of hypermedia systems structured with static links is a nightmare that makes the technique unworkable in dynamic environments.

An extensible link engine is another approach to manage the navigation relations. The link engine is the part of the hypermedia system that determines what targets can be reached from a given source. An extensible link engine is a link engine that is able to incorporate domain specific algorithms to infer relationships between information fragments. With an extensible link engine, the hypermedia system is able to incorporate knowledge about the information domain and to help in authoring and maintaining the navigation relations.

EXAMPLE

> Hypermedia technology has been successfully applied in many technical documentation projects, so it should not come as a surprise to recommend a hypermedia system as an ideal medium to explore framework structures. However, compared with traditional technical documentation projects, framework structures demands an additional requirement from the underlying hypermedia system: flexible adaptation towards changes. A concrete framework applied in real life situations is constantly evolving. On the one hand, the framework is incarnated in applications part of the framework's target design space. On the other hand, the framework is redesigned to cope with new requirements. Both have impact on the framework structure and the hypermedia system should handle such structural changes effectively. To provide adequate support, the hypermedia system must incorporate knowledge from the programming environment to infer information structures automatically.

As an example, consider the links connecting a method to all senders (implementors) of the corresponding message. One cannot expect a software engineer to maintain such links manually: they should be inferred from the compiler part of the framework development environment. Moreover, to offer true support in managing frameworks, the links connecting a framework element to the design patterns it participates in should be maintained within the framework development environment as well.

Problem

How can one incorporate domain specific link resolution algorithms in the previously defined navigation model (see [navigation]), so that resolution algorithms can be maintained easily ?

Solution

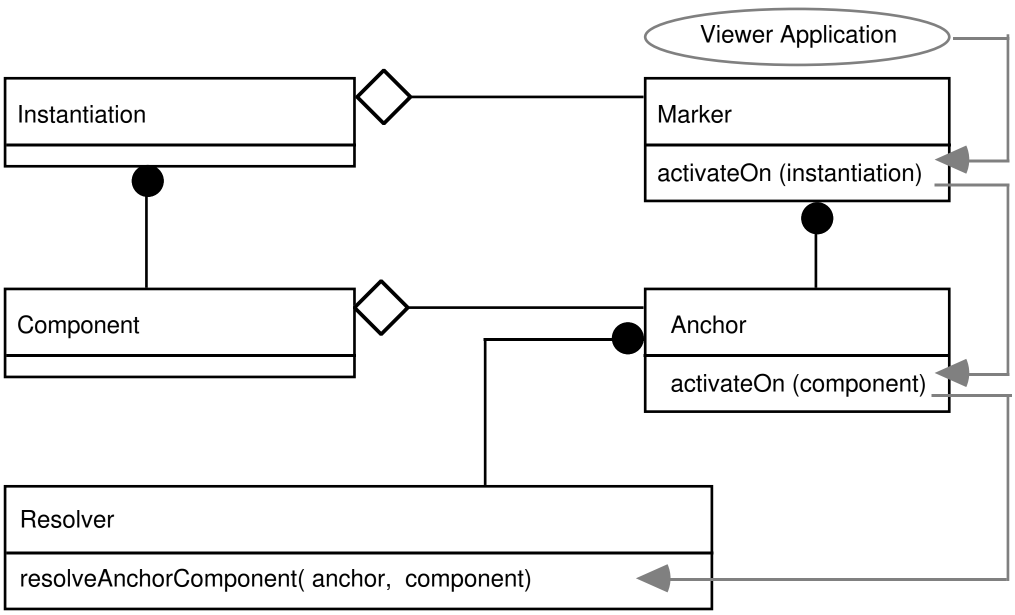
Create a new class of objects, where each object represents a link resolution algorithm. Call such an object a *resolver*. Define a contract on markers and anchors, stating that all navigation actions are passed to such a resolver to determine the appropriate targets.

EXAMPLE

The Zypher framework browser incorporates navigation actions that bring a software engineer from a given message selector to all the places where such a message is implemented and all the places where such a message is sent. This counts for two resolvers: a senders resolver and an implementors resolver. Also there are navigation actions that bring a software engineer from a framework element (a class or a method) to a design pattern where this framework element plays a particular role. Determining these target patterns is accomplished by the pattern resolver.

The Zypher framework browser also incorporates a resolver for every URL address space. Note that some special address spaces for the framework browser domain are added (browser, pattern, name).

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Component

A component passed as a parameter to the `activateOn` method defined on Anchor.

Instantiation

An instantiation is passed as a parameter to the `activateOn` method defined on Marker.

Marker

All markers implement the `activateOn` message sent by the viewer application to start a navigation action. Markers receiving this message, must pass it on to the anchor they represent, but are allowed to perform additional actions.

Anchor

All anchors implement the `activateOn` message sent by a marker to start a navigation action. Anchors receiving this message, must pass it on to the resolver by means of the `resolveAnchorComponent` message, but are allowed to perform additional actions.

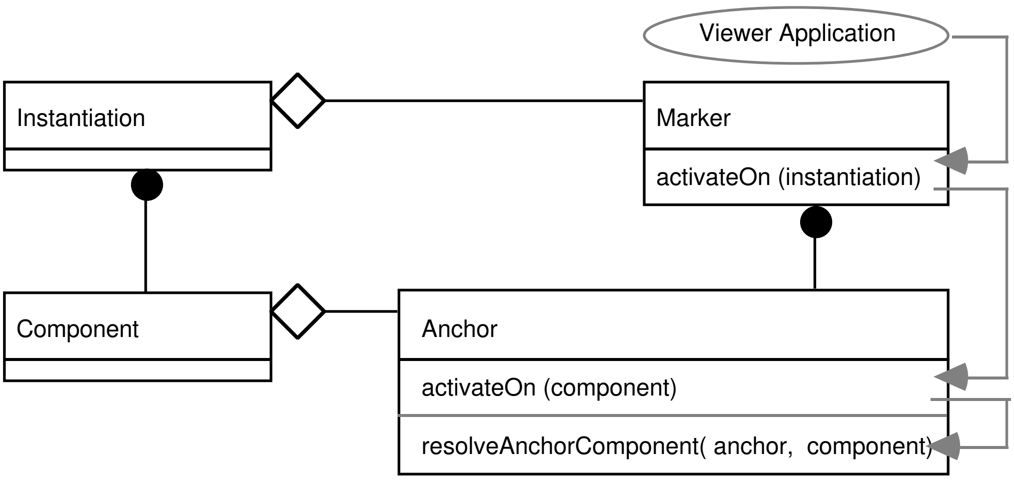
Resolver

A resolver models a specific link resolution algorithm. A resolver implements the `resolveAnchorComponent` message sent by the anchor to determine the target of a navigation action. Specific resolver classes are allowed to extend the message set for a finer grained interaction with the anchor.

Motivation

The determination of the navigation targets is crucial in all hypermedia systems. The need to incorporate domain specific knowledge forces us to make the determination process explicit in a resolver method. Why should this method be defined on a separate class, why not use an existing one ?

The most obvious candidate would be the Anchor class, which would result in the following structure.



In the above schema, an anchor captures two different aspects of the design, namely the structural reference to a substructure of a component and the way to resolve an anchor-component pair. Mixing two different aspects in a single design element causes difficulties as the design is less configurable and less extensible.

a) Configurability

Some anchors may resolve differently depending on the navigation context, which means that in some occasions the resolver process must change at run-time. Changing the resolver process is not possible if the two are captured in a single design element. Moreover, decoupling the resolver from the anchor enables lazy structures, one can defer to choice of the resolution algorithm until it is needed.

EXAMPLE

The Zypher framework browser incorporates navigation actions that bring a software engineer from a given message selector to all the places where such a message is implemented and all the places where such a message is sent. Thus the same anchor (the message selector) is used in different resolution algorithms: one wants to attach the resolver to the anchor depending on the particular action that triggers the navigation.

Zypher framework documentation is stored inside HTML-documents and Microsoft Word documents and contain many anchors in URL format. In the URL format, the anchor value starts with a keyword defining the URL address space, followed by some address in a format depending on the address space. The keyword identifies the resolver that is supposed to interpret the address. However, it is not a very good idea to parse and interpret all keywords in all anchors to determine the resolver. Most of the anchors available in the document are not activated, so it is better to wait until the actual activation to interpret the keyword.

b) Extensibility

Like argued in the [interoperability] pattern, separating the reference aspect from the resolver aspect reduces the number of classes by factoring out the behaviour along the two dimensions.

EXAMPLE

Message selectors and class names can be found in lists (i.e. the list panes of the pattern browsers and code browser) in which case an index anchor is used, or they can be found in Smalltalk source code in which case a range anchor is used. On both classes of anchors the senders, implementors and pattern resolver may be applied.

Both HTML-documents and Microsoft Word documents contain many anchors in URL format, referring to various address spaces. Anchors in a HTML-document hold a range to refer inside the document, while anchors for Microsoft Word documents use bookmarks (i.e. value markers).

As can be deduced from the table in [figure 19], we reduce the number of classes to twelve (= 9 + 3) instead of eighteen (= (3 * 2) + (6 * 2)). Moreover, all URL resolver classes are defined as a subclass from a common abstract URL resolver, which makes it very easy to extend the system with new URL address spaces.

Anchor \ Resolver	Index	Range	Value
Senders	x	x	
Implementors	x	x	
Pattern	x	x	
URL		x	x
URL File		x	x
URL HTTP		x	x
URL Browser		x	x
URL Pattern		x	x
URL Name		x	x

Figure 19: Combinations of anchors and resolvers

Issues

Naming

The concept of a resolver as defined here is not part of the Dexter model [Halasz,Schwartz'90]. However, the Dexter specification includes the notion of a resolver function (Dexter is quite vague there), which explains the choice of the name.

Structural Objects versus Behavioural Objects

It was not a coincidence that the same arguments (i.e. configurability and extensibility) that plead for a separation between components, anchors, instantiations and markers, motivate the existence of a separate resolver object. However, the absence of the dynamics argument, suggests that there is another factorisation technique at work here.

The difference between the factorisation applied in the [interoperability] and [navigation] patterns and the factorisation applied here is based on the difference between structure and behaviour.

To understand why, note that the component, anchor, instantiation and marker objects represent elements for data structures: they must be assembled properly to model external information. The resolver object, however, represents a process that interprets a particular constellation of these elements. The resolver thus assigns behaviour to a structure, which explains the terms structural object and behavioural object.

Consequences

If the configuration issue was important when discussing structural objects (see [interoperability ~ consequences], [navigation ~ consequences]), then it is even more important if a design incorporates behavioural objects. After all, the decision about what behavioural object should handle what structural objects has a great impact on the global behaviour of the system. The issue is dealt with in the [meta-meta-objects] design pattern.

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned how the introduction of the resolver object allows to incorporate domain specific algorithms that determine navigation relations. The obvious next step is then the [editor] pattern, that models a viewer application.

The [meta-meta-objects] pattern determines what resolver should handle a given anchor/component pair.

The resolver plays an important role in the navigation template as described in the [navigation template].

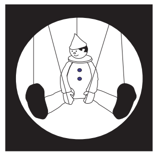
Other Catalogues

The messages defined in the contracts section participate in the template method pattern as defined in [GammaEtAl'93] and [Pree'94]. To use the terminology of the latter, the `resolveAnchorComponent` method defined on resolver is a hook method for the `activateOn` template method defined on anchor. In turn, the `activateOn` method defined on anchor is a hook method for the `activateOn` template method defined on marker. Both are forms of the 1:1 connection meta-pattern. The `activateOn` and `resolveAnchorComponent` messages are hook methods for the navigation template [navigation template].

Besides participating in the 1:1 connection meta-pattern, the `activateOn` messages defined on anchor and marker participate in the 1:1 recursive connection meta-pattern, meaning that subclasses overriding the `valueOn` message must include the super method in their behaviour.

Editor: Incorporate External Viewer Applications

Intent



Encapsulate the command sets that control the viewer applications presenting information and participating in the navigation.

Analysis

Hypermedia is about managing and presenting information. A crucial part of all hypermedia systems is the module that is responsible for presenting information. Presenting information in a hypermedia system is concerned with presenting a particular information fragment, highlighting substructures of an information fragment as candidate source of a navigation operation and selecting a substructure of an information fragment as target of a navigation operation.

A complicating factor is the notion of [interoperability], where the hypermedia system is able to incorporate third party applications to present information to the user and third party applications can request link services from the hypermedia system. Moreover, the same information can be presented by different viewer applications. Note that these applications are allowed to modify both the information and the navigation structures.

Problem

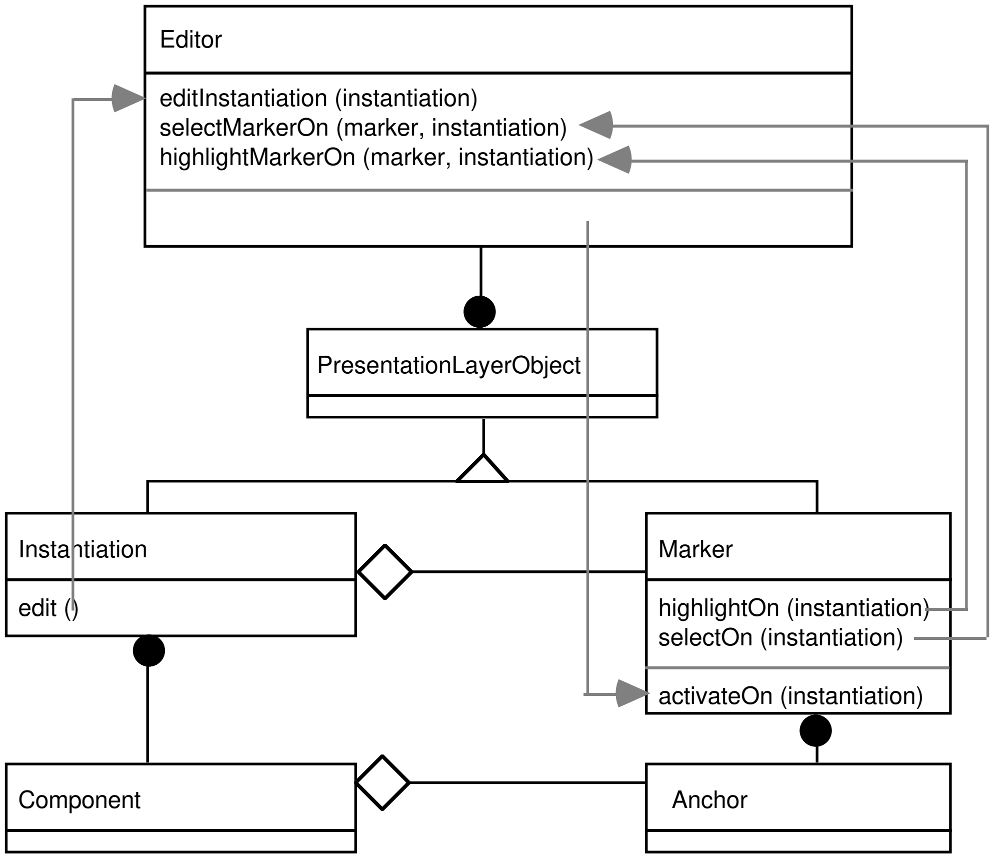
How can one model an interoperable information system where the same information can be manipulated by different presentation applications, possibly not part of the hypermedia system ?

Solution

Create a special object that represents an information presentation application. Call this object an *editor*, to emphasise the idea that these applications can modify information.

Create a special superclass for all objects that can be manipulated by an editor object; call this a *PresentationLayerObject*.

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Editor

An editor object represents a module of the hypermedia system (possibly an interface to a third party application) that can be asked to present information.

All editors implement the `editInstantiation` method called by an instantiation object that wants to be presented. All editors implement the `highlightMarker` method called by a marker object that wants to be highlighted as a candidate source for a navigation operation. All editors implement the `selectMarker` method called by a marker object that wants to be selected as a target for a navigation operation.

PresentationLayerObject

A presentation layer object is an abstract superclass for all objects that can be manipulated by an editor object.

Instantiation

All instantiations implement the `edit` method called by whatever object that wants to present the instantiation. This method is delegated to an `editInstantiation` method defined on an editor associated with the instantiation.

Marker

All markers implement the `highlightOn` method called by whatever object that wants to highlight the marker as a source of a navigation operation. This method is delegated to an `highlightMarkerOn` method defined on an editor associated with the marker.

All markers implement the `selectOn` method called by whatever object that wants to select the marker as a target of a navigation operation. This method is delegated to an `selectMarkerOn` method defined on an editor associated with the marker.

All markers implement the `activateOn` method called by an editor object when it detects that the user has activated this marker as a source of a navigation operation.

Motivation

Why should the `edit, highlightOn, selectOn` methods be defined on a separate class. The same arguments that made us separate the resolver from the anchor (see [resolver]) apply (i.e. configurability and extensibility).

Consequences

If the configuration issue was important when discussing structural objects (see [interoperability ~ consequences], [navigation ~ consequences]), then it is even more important if a design incorporates behavioural objects. After all, the decision about what behavioural object should deals with what structural objects has a great impact on the global behaviour of the system. The issue is dealt with in the [meta-meta-objects] design pattern.

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned how the introduction of the editor object allows to incorporate specialised viewer application. The obvious next step is then the [loader] pattern, that models an information repository.

The [meta-meta-objects] pattern determines what editor should handle a given instantiation or marker.

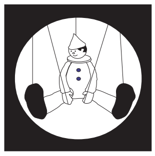
The editor plays an important role in the navigation template as described in the [navigation template].

Other Catalogues

The messages defined in the contracts section participate in the template method pattern as defined in [GammaEtAl'93] and [Pree'94]. To use the terminology of the latter, the `editInstantiation, selectMarkerOn, highlightMarkerOn` methods defined on editor are hook methods for the `edit` template method defined on instantiation and the `highlightOn`, `selectOn` template methods defined on marker. In turn, the `activateOn` method defined on marker is a hook method for an unspecified template method defined on editor. Both are forms of the 1:1 connection meta-pattern. The `edit, highlightOn` and `selectOn` messages are hook methods for the navigation template [navigation template].

Loader: Incorporate Information Repositories

Intent



Encapsulate the commands that control the repositories storing the information.

Analysis

Hypermedia is about managing and presenting information. A crucial part of all hypermedia systems is the module that is responsible for storing and retrieving information from information repositories.

A complicating factor is the notion of [interoperability], where the hypermedia system must be able to incorporate various information repositories outside the hypermedia system. This implies that the information can be changed from outside the hypermedia system.

Problem

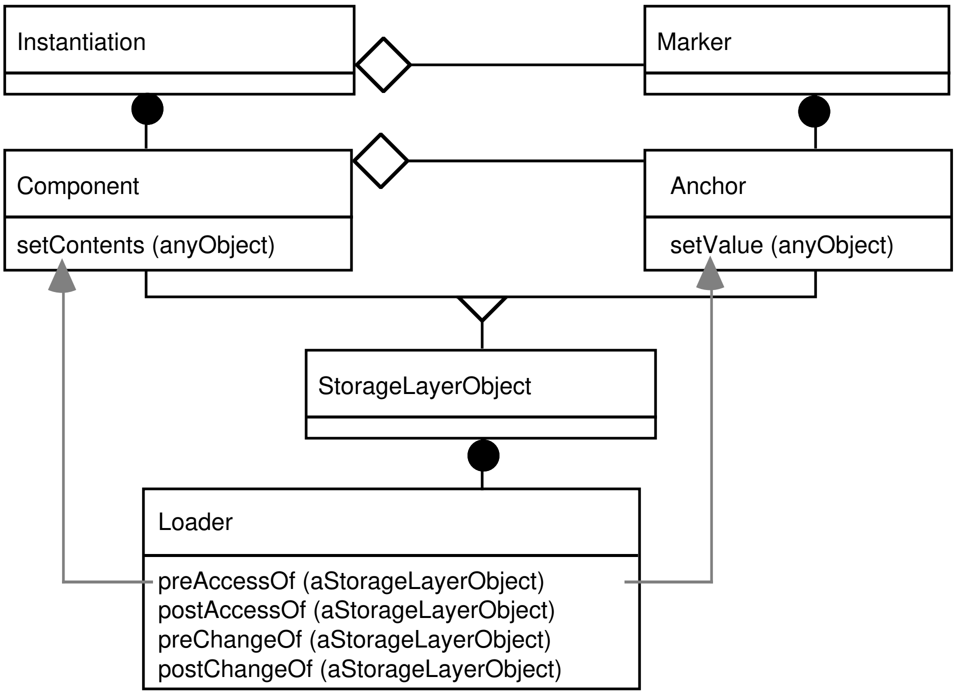
How can one model an interoperable information system where information can be supplied from different repositories, not part of the hypermedia system ? How can one deal with the problem that information can be modified from outside as from within the system ?

Solution

Create a special object that represents an information repository. Call this object a *loader*.

Create a special superclass for all objects that can be manipulated by a loader object; call this a *StorageLayerObject*.

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Loader

A loader object represents a module of the hypermedia system (possibly an interface to a third party application) that can be asked to store or retrieve information.

All loaders implement the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` messages, sent whatever object that accesses or changes a storage layer object. The loader then checks whether the contents of the storage layer object corresponds with what is in the repository and adapts the contents when necessary.

StorageLayerObject

A Storage layer object is an abstract superclass for all objects that can be manipulated by an editor object.

Component

All components implement the `setContents` method called by the loader when it detects that the contents of a component does not correspond with the contents in the repository.

Anchor

All anchors implement the `setValue` method called by the loader when it detects that the contents of an anchor does not correspond with the contents in the repository.

Motivation

Why should the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` methods be defined on a separate class. The same arguments that made us separate the resolver from the anchor (see [resolver]) apply (i.e. configurability and extensibility).

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned how the introduction of the loader object allows to incorporate specialised repositories. The obvious next step is then the [evens] pattern, that imposes a layered structure on top of the Zypher framework and introduces event passing as a mechanism to transfer results from lower layers to upper layers.

The [meta-meta-objects] pattern determines what loader should handle a given component or anchor.

The loader plays an important role in the navigation template as described in the [navigation template].

Other Catalogues

The messages defined in the contracts section participate in the template method pattern as defined in [GammaEtAl'93] and [Pree'94]. To use the terminology of the latter, the `setContents` method defined on component and the `setValue` method defined on anchor is a hook method for an unspecified template method defined on loader. This is a form of the 1:1 connection meta-pattern.

CHAPTER *6*

A Layered Hypermedia Framework

This part of the dissertation imposes a layered structure on top of the Zypher framework and introduces event passing as a mechanism to transfer results from lower layers to upper layers. The design pattern part of this chapter is not captured in the framework design methodology proposed in the Zypher contribution (p.63).

Layered Structure

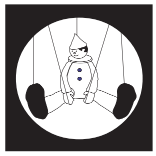
Layering is frequently applied in software engineering, although the term is vague and seldom refers to a unique technique. Frameworks are organised into layers to release the tension between ease of use and extensibility. According to [Kiczales,Rivières,Bobrow'91], a complex object-oriented structure is split into modules, requiring that the main module delegates functionality to subsidiary modules to do parts of the overall task. Then, even though each module supports extensibility along only a single dimension, the combination supports extensibility along multiple dimensions. The upper modules tend to provide more powerful access than the lower modules, but with this power comes more responsibility, which makes it harder to extend the system. Lower modules tend to be more focused and easier to extend, but have lesser impact on the overall behaviour

The notion of layering exists in hypermedia research as well, among others in the Hypertext Abstract Machine [Campbell,Goodman'87] and the Dexter model [Halasz,Schwartz'90]. There, layering releases the same tension between ease of use and extensibility, although there is a slight shift of focus as layers in object-oriented frameworks tend to emphasise on different levels of abstraction, while layers in hypermedia systems are more oriented towards different levels of functionality.

Nevertheless, a prominent property of all layered structures is that lower layers are not supposed to communicate directly with upper layers, because this violates abstraction barriers. Event passing is a technique to circumvent abstraction barriers properly.

Events: Co-ordinate Functional Layers

Intent



Co-ordinate the behaviour of the modules constituting the framework, while avoiding the exhibition of implementation details and respecting the unawareness constraint.

Analysis

Elsewhere ([interoperability], [navigation], [resolver], [editor], [loader]) we have defined the protocol between the different modules constituting the Zypher framework. Basically, we have been factoring out responsibilities in different modules, guided by the insight that separating responsibilities lead to better *configurability* (i.e. instead of constructing all possible configurations in advance and choosing the appropriate one, we can defer the decision and assemble the desired configuration at run-time) and better *extensibility* (because the modules focus on a single aspect it is easier to inherit behaviour and sometimes it reduces the number of classes). This resulted in the following structure (we left out the messages to avoid clutter).

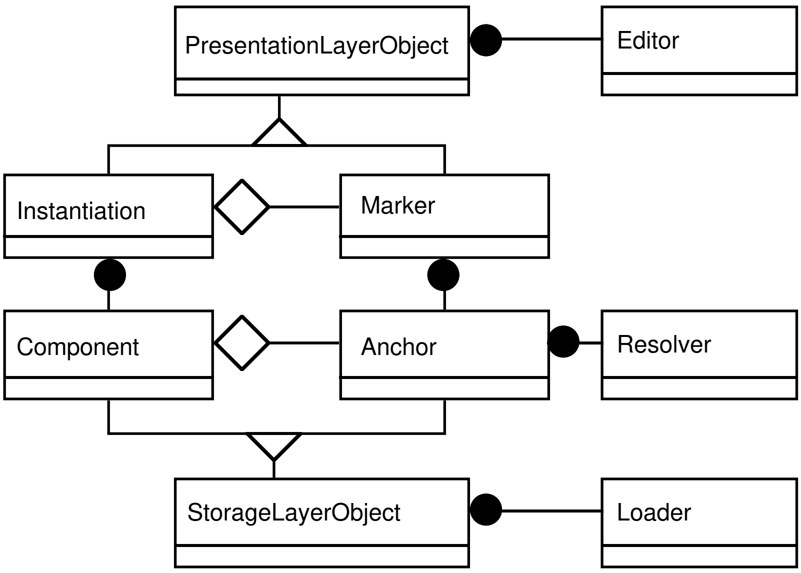


Figure 20: Overview of the Structure

All these modules may exchange messages, which raises the issue of complexity. Since there are nine parties involved, the number of possible communication patterns is quite large and so the resulting hypermedia system is quite complex. To solve this, it is common practice in the hypermedia community (see the Hypertext Abstract Machine [Campbell,Goodman'87] and the Dexter model [Halasz,Schwartz'90]) to organise the modules into three *layers* according to the principal functionalities in the hypermedia system. Each layer introduces a level of abstraction: modules in the lower layers are not allowed to send requests to modules in the upper layers, but sending requests from upper levels to lower levels is permitted. This principle of one way communication between layers reduces the number of possible communication patterns and thus the complexity of the hypermedia system. The layered structure imposes the *unawareness constraint* defined in [interoperability] and [navigation].

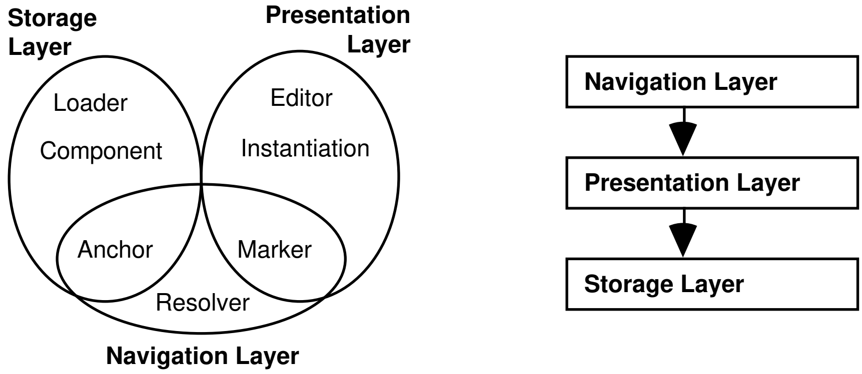


Figure 21: Layered Structure

The Zypher Hypermedia Framework is organised around three layers: storage, presentation and navigation. The *storage layer* provides the necessary elements to maintain the persistent part of a hypermedia structure. The location and format of hypermedia information is encapsulated, as is the command language needed to conduct external storage devices. The *presentation layer* is responsible for the run-time behaviour of a hypermedia system and provides elements to present and modify information fragments. The presentation layer encapsulates the control language to instruct different information viewers to perform the desired behaviour. To attain this functionality, the presentation layer co-ordinates with the storage layer. The *navigation layer* provides elements to model the endpoints that serve as the source and/or targets of navigation operations. The decision process of what endpoints are connected to each other is encapsulated, as is the run-time behaviour of the navigation operation itself. To attain this functionality, the navigation layer co-ordinates with the storage and presentation layer.

The principle of one way communication between layers implies that the navigation layer is allowed to send requests to both the presentation layer and storage layer; the presentation layer is allowed to send requests to the storage layer but not to the navigation layer and the storage layer is forbidden to send requests to other layers. The different contracts as they are specified so far ([interoperability], [navigation], [resolver], [editor], [loader]), respect this principle because of the unawareness constraint. However, there is an important aspect of co-operation that is not yet covered in the contracts namely the way the different modules synchronise their internal state.

EXAMPLE

> To modify the framework documentation in HTML-format, a software engineer opens a text editor and a HTML-browser on the same file; the text editor is used to modify the contents in HTML-format and the HTML-browser is used to check the layout of the document. When the text editor saves a new version of the documentation into the file, the HTML-browser (and all associated data structures) should synchronise their internal state.
>
> Almost all code and pattern browsers maintain a list of the messages sent by a particular method manipulated in a source code pane. If the source code pane modifies the source of a method then the list of messages must be updated.

Of course, specific modules are supposed to add specific behaviour and thus are allowed to refine the contracts. One such refinement could be the synchronisation of internal state. However, if one delegates behaviour to specific modules, one risks that the modules make assumptions about the other contract counterparts and as a result have limited applicability.

The modules would be tightly coupled and especially with synchronisation this would imply that the extensibility argument for separating responsibilities is not valid anymore. So we need to specify contracts to co-ordinate the behaviour of the modules and we must ensure that no implementation details are exhibited in these contracts.

EXAMPLE

In the above example of coherent edition of HTML-files, we could adapt the text editor and HTML-browser so that when a new version is saved all associated HTML-browsers are enumerated and commanded to update the appropriate elements. However, this would imply that this text editor only works in combination with HTML-browsers and not with any other browser based on text files.

To synchronise the list of messages sent by a particular method with the pane containing the source code of that method, we could modify the source code pane and list pane to synchronise their contents. Nevertheless, this would imply that the list and source code pane are privileged partners that cannot co-ordinate with other panes.

Problem

How to specify contracts to co-ordinate the behaviour of the modules with the constraints that no implementation details should be exhibited and that the layered architecture (storage, presentation and navigation) must be preserved ?

Solution

Co-ordinate the behaviour that respects the unawareness constraint by means of *event passing*. Instead of adding complicated synchronisation protocols, the contracts are extended with simple event passing mechanisms and implement the actual synchronisation via existing contracts. In other words, rather than forcing a module to send messages to all other modules that should synchronise, make the original module broadcast an event to warn interested modules. On receiving such an event, interested modules may adapt to the new situation using the existing interface applied in the original contract.

Design the event protocol to focus on the important events in an object's life cycle (i.e. creations, destruction's, updates, access) to avoid the exhibition of implementation details. Organise this protocol in three categories (navigation, presentation and storage) according to the three layers (see [figure 21]).

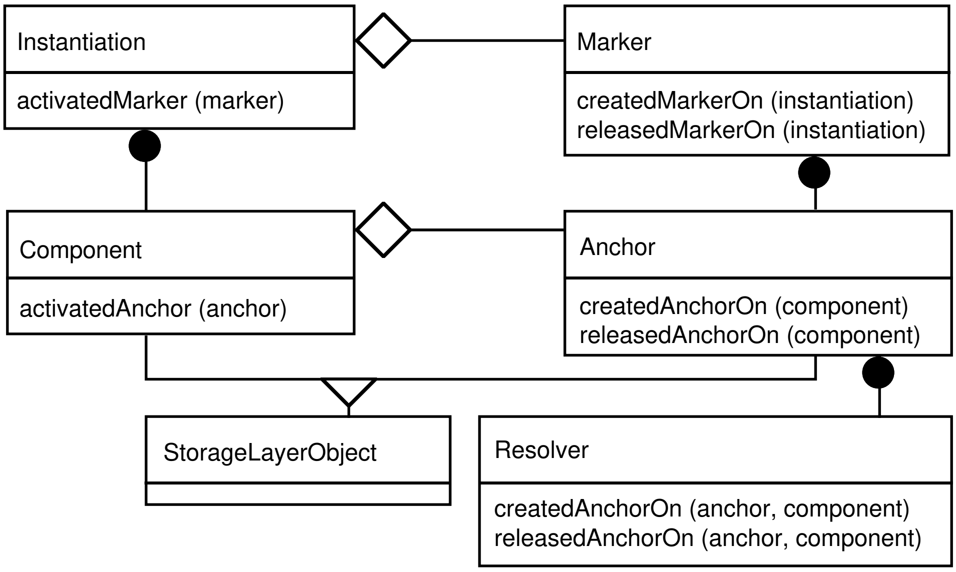
EXAMPLE

To synchronise the text editor and HTML-browser, both viewer applications should have an instantiation working on the same file component; the text editor is used to modify the contents in HTML-format, while the HTML-browser is used to inspect the layout of the document. When the text editor modifies the contents of the file component, an update event is broadcasted to all associated anchors and instantiations. Receiving the update event, the anchors set a dirty bit and pass the event on to their resolver that is able the clean up auxiliary data structures (i.e. cached results). When the HTML-instantiation receives an update event, it parses the contents of the file component and instructs the HTML-browser to display the adjusted layout. This parsing process may affect the anchors as well: positions of anchors may be adjusted, new anchors may be created and others may be released. When the position of an anchor is adjusted, an update event is broadcasted to the associated markers to allow an update of the display. When an anchor is created or released, an event is broadcasted to all interested resolvers in order to set-up or remove auxiliary data structures.

To keep the source code pane and the list pane of code and pattern browsers coherent, both panes should have an instantiation working on the same Smalltalk component. When the source code pane modifies the contents of the Smalltalk component, an update event is broadcasted to all associated anchors and instantiations, among others the list pane and the marker containing the current selection inside the list pane. The list pane requests the new message list from the Smalltalk component and adjusts its display; the marker is reset.

Contract

a) Navigation



(See [class diagrams] for a short survey of the main elements in a class diagram)

Instantiation

All instantiations implement an `activatedMarker` message sent by whatever object that activates a navigation action on a particular instantiation. This allows the instantiation to adjust its internal state after a navigation operation was completed. See [meta-objects ~ contracts] for further details.

Marker

All markers implement the `createdMarkerOn`, `releasedMarkerOn` messages sent by whatever object that alters the aggregation relation between an instantiation and a marker. This allows the marker to adjust its internal state after the alteration of the aggregation relation was completed. See [meta-objects ~ contracts] for further details.

Component

All components implement an `activatedAnchor` message sent by whatever object that activates a navigation action on a particular component. This allows the component to adjust its internal state after a navigation was completed. See [meta-objects ~ contracts] for further details.

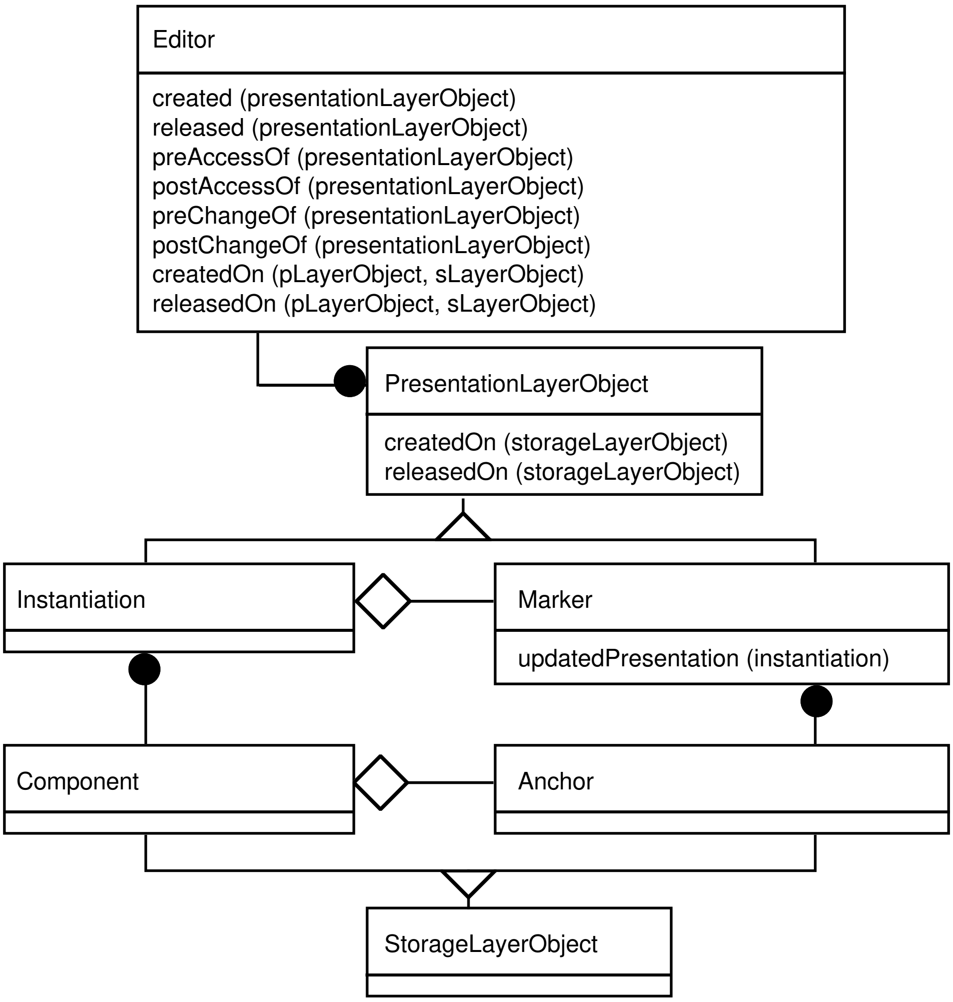
Anchor

All anchors implement the `createdAnchorOn`, `releasedAnchorOn` messages sent by whatever object that alters the aggregation relation between a component and an anchor. This allows the anchor to adjust its internal state after the alteration of the aggregation relation was completed. See [meta-objects ~ contracts] for further details.

Resolver

All resolvers implement the `createdAnchorOn`, `releasedAnchorOn` messages sent by whatever object that alters the aggregation relation between a component and an anchor. This allows the resolver to adjust its internal state after the alteration of the aggregation relation was completed. See [meta-objects ~ contracts] for further details. See [meta-objects ~ contracts] for further details.

b) Presentation



(See [class diagrams] for a short survey of the main elements in a class diagram)

PresentationLayerObject

All markers and instantiations implement the `createdOn`, `releasedOn` messages sent whatever object that alters the association between an instantiation (marker) and a component (anchor). This allows the instantiation (marker) to adjust its internal state after the alteration. See [meta-objects ~ contracts] for further details.

Marker

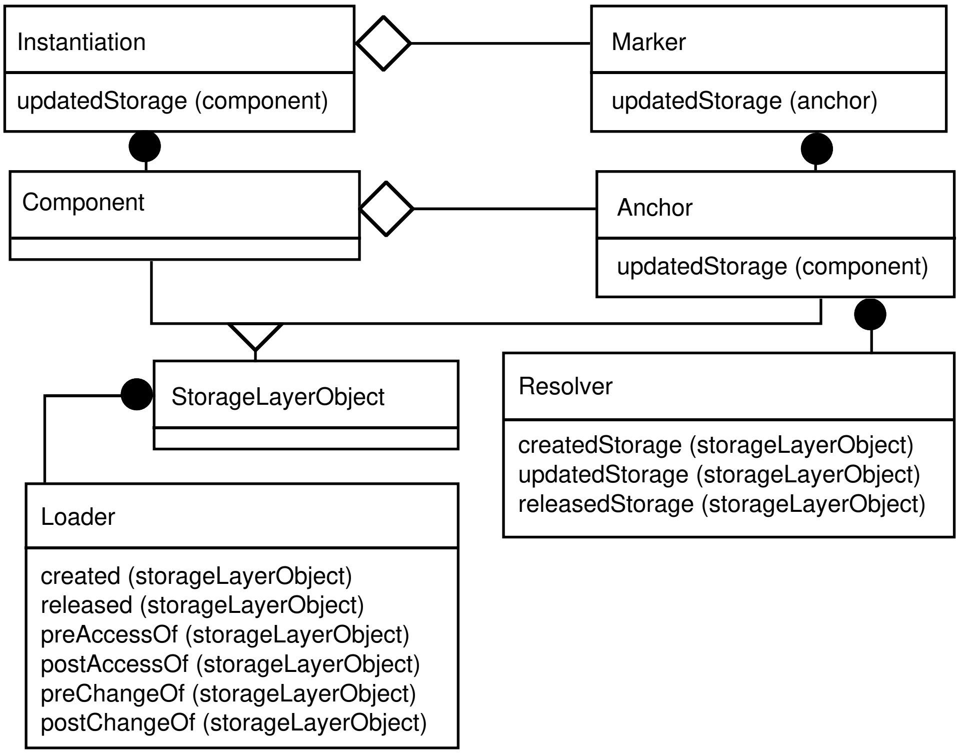
All markers implement an `updatedPresentation` message sent by whatever object that alters an instantiation associated with that marker. This allows the marker to adjust its internal state after the instantiation was altered. See [meta-objects ~ contracts] for further details.

Editor

All editors implement the `created`, `released`, `preAccessOf`, `postAccessOf`, `preChangeOf`, `postChangeOf` message sent by whatever object that alters a presentation layer object. This allows the editor to adjust its internal state, or to adjust the internal state of the presentation layer object after the alteration. See [meta-objects ~ contracts] for further details.

All editors implement the `createdOn`, `releasedOn` messages sent by whatever object that alters the association between an instantiation (marker) and a component (anchor). This allows the editor to adjust its internal state after the alteration. See [meta-objects ~ contracts] for further details.

c) Storage



(See [class diagrams] for a short survey of the main elements in a class diagram)

Instantiation

All instantiations implement an `updatedStorage` message sent by whatever object that alters a component associated with that instantiation. This allows the instantiation to adjust its internal state after the component was altered. See [meta-objects ~ contracts] for further details.

Marker

All markers implement an `updatedStorage` message sent by whatever object that alters an anchor associated with that marker. This allows the marker to adjust its internal state after the anchor was altered. See [meta-objects ~ contracts] for further details.

Anchor

All anchors implement an `updatedStorage` message sent by whatever object that alters a component associated with that anchor. This allows the anchor to adjust its internal state after the component was altered. See [meta-objects ~ contracts] for further details.

Loader

All loaders implement the `created`, `released`, `preAccessOf`, `postAccessOf`, `preChangeOf`, `postChangeOf` message sent by whatever object that alters a storage layer object. This allows the loader to adjust its internal state, or to adapt the internal state of the storage layer object after the alteration. See [meta-objects ~ contracts] for further details.

Resolver

All resolvers implement the `createdStorage`, `updatedStorage`, `releasedStorage` messages sent by whatever object that alters a storage layer object. This allows the resolver to adjust its internal state after the alteration of the storage layer object was completed. See [meta-objects ~ contracts] for further details.

Motivation

The point of the notification mechanism was the need for a synchronisation mechanism between different modules without one module sending messages to the other. Why is the proposed event mechanism a good solution for this problem ?

a) Abstract Coupling

The main motivation is that the event mechanism provides an *abstract coupling* between modules. Call the *master* module the module that is the origin for the event, and call the *slave* modules all the modules that should synchronise with the master. The coupling between the master and the slave is abstract since the master does not know the number of slave modules that are interested, nor does the master module know the interface of those slave modules. The master communicates with the slaves based on abstract events like creations, updates, reads and releases. The coupling between the different slave modules is abstract as well, since different slave modules are not aware of each other.

EXAMPLE

When the content of the file component is modified, all associated instantiated anchors and instantiations are notified through the `updatedPresentation` message. The file component (i.e. the master) does not have to know how many of associated objects there are, nor is it necessary to be aware of the class of these associated objects. The HTML and text instantiation (i.e. the slaves) do not have to communicate with each other.

A Smalltalk component that has been modified results in a notification of all associated instantiations (panes) through the `updatedPresentation` message. The Smalltalk component (i.e. the master) does not have to bother about the number of associated panes and the class of these panes. The list and source code panes (i.e. the slaves) are completely unaware of each other.

b) Minimal Interface

Note that the relation between a slave module and a master module is not abstract, since the slave is aware of the master module and must know the master's interface. This is not a problem, since this is part of the normal contract between the slave and the master. Which brings us to the second motivation, the fact that the event mechanism allows for a *minimal interface* between modules. Implementors of modules accept that they should provide messages to synchronise their activities, but they want this interface to be a minor part of the global implementation since this requires less development effort and is easier to maintain. The interface between two modules is based on abstract events like creations, updates, reads and releases and the rest of the synchronisation is based on the normal co-operation needed to implement the hypermedia system itself.

EXAMPLE

When the HTML-instantiation receives an `updatedStorage` message, its content is updated by parsing the HTML-stream contained in the file component. The parsing algorithm is already part of the instantation's implementation since it is essential for the display of a HTML-document.

A list pane receiving an `updatedStorage` message updates the display just as it fills up the display the first time it is opened: it requests the component for its new content and enumerates all items in the list to pass them to the display routine.

c) Respect Layered Structure

Finally, the event mechanism respects the unawareness constraint imposed by the layered structure. To explain this, it is important to understand that master modules always are the objects in the lower layers. When the master module detects a certain event, it warns a global dispatching object (see [meta-objects]) which in turn notifies all the interested modules in the upper layers. Since a master module never calls upon the services of its slaves directly, and since the master modules always belong to the lower layers, the layered structure is not violated.

Consequences

There are two important questions left unanswered here. One is what objects sends the notification messages, and the other is what is the central dispatching object ? We postpone the answers to these questions until [meta-objects] and [meta-meta-objects].

Relations

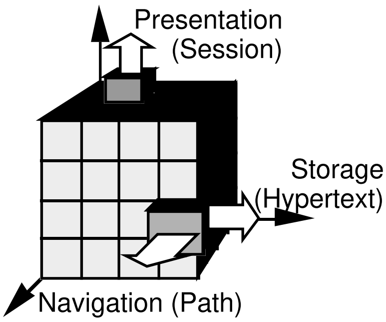
Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned about the layering structure in the Zypher framework and how the introduction of event allows to transfer results from lower layers to upper layers. The obvious next step is then the [meta-objects] pattern, that provide an interface for plugging services that affect the global behaviour of the hypermedia system.

Other Catalogues

This pattern is a superposition of a number of observer patterns as defined in [GammaEtAl'93].

Tailorability In An Open Hypermedia Framework



This chapter introduces three meta-objects in the design of the Zypher Open Hypermedia Framework —Path, Session, and Hypertext— and one meta-meta-object HypermediaContext.

The introduction of the meta-objects is a necessary step to introduce tailorability in the design space for hypermedia systems, this way turning a interoperable hypermedia system into a completely open hypermedia system. A design space is delineated by three orthogonal design axes, where each axis represents aspects important for all hypermedia systems, namely navigation, presentation and storage. A hypermedia system is then a three dimensional volume in that design space. The meta-objects are chosen to represent one of these design axes (i.e. Path for the navigation axis, Session for the presentation axis and Hypertext for the storage axis), while the meta-meta-object (HypermediaContext) represents the axis system itself.

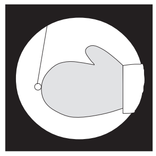
As such, the meta-objects are introduced as explicit representations for the framework contracts that follow from the third of our framework design guidelines, stating that "Each design space axis should correspond to a framework contract". The one meta-meta-object is introduced as an explicit representation for the framework contract that followed from the fourth of our framework design guidelines, stating that "The configuration of the design space axes should correspond to a framework contract".

The purpose of these meta-levels is the following. The design patterns in the previous chapters describe the base level of a hypermedia framework. This base level introduces domain level tailorability, i.e. the ability to add points to an axis in the design space. The introduction of the three meta-objects —Path, Session, Hypertext— extends the framework with system level tailorability, i.e. the ability to change the behaviour of all points residing on at least one axis in a uniform way. Finally, the introduction of the HypermediaContext meta-meta-object extends the framework with configuration level tailorability, i.e. the ability to change the relationship (in the mathematical sense of the world) between points residing in different axes of the design space.

The three levels of tailorability are recapitulated in this chapter to emphasise on the different roles they perform in the framework design.

Meta-objects: Introduce System Level Tailorability

Intent

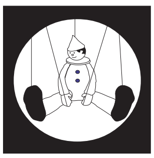


Provide an interface for plugging services that affect the global behaviour of the hypermedia system.

Analysis

Because a hypermedia system encourages the exploration of information (i.e. see [navigation]), the approach is a potential benefit for all computer applications that deal with information. But it is impossible to build a hypermedia system that incorporates all functionality required by all possible application domains. Therefore, a hypermedia system designer requires a *tailorable* hypermedia framework that enables the installation of special features needed for the application domain at hand.

In the case of the Zypher hypermedia framework, a hypermedia system designer may assemble a hypermedia system for a particular application domain by creating specific resolver-, editor-, loader-, anchor-, marker-, instantiation- and component classes and glueing them all together. This corresponds with the notion of *domain level tailorability*.



Domain Level Tailorability

(The icon associated with this tailorability level is based on the puppet master metaphor (see [puppet master metaphor]).

Domain level tailorability is the kind of tailorability needed to deliver a hypermedia system for a specific application domain.

Domain level tailorability is achieved by extending the basic hypermedia framework with domain specific classes. Creating such domain specific classes requires a great deal of technical expertise about the software systems applied in the problem domain but has little to do with the hypermedia system as such. One does not need to understand the inner details of the hypermedia system to tailor the system. Note that some classes, if written 'good', can be reused for different problem domains.

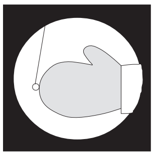
Typical usage of domain level tailorability in hypermedia systems would involve the incorporation of new data formats, extra information presentation methods or supplementary types of navigation techniques. *Domain level tailorability corresponds with the addition of extra points (or the modification of existing points) on the axes of the Zypher design space.*

EXAMPLE

The Zypher framework documentation is stored in files or on a HTTP server. This accounts for storage layer classes like the file and HTTP component and the file and HTTP loaders. To manipulate this documentation using the Microsoft Word third party application or the home cooked text editor and the HTML-browser, one must add presentation layer classes like the Microsoft Word, HTML and Text instantiations and editors. To allow navigation between documents, the navigation layer is equipped with a range and value anchor; sensitive text and active bookmark markers and several URL resolvers.

The various code and pattern browsers that inspect the design and implementation of the framework fetch their information from the Smalltalk development environment and a pattern fact base. This explains storage layer classes like the class, method and pattern fact components and the Smalltalk and pattern loader. The browsers themselves are assembled from pre-compiled parts like the list pane, the source code pane. The navigation is made possible by index anchors, range anchors, menu markers, list markers, Smalltalk markers, Smalltalk resolvers and pattern resolvers.

However, experience has shown that system designers are not satisfied with the potential of domain level tailorability. Sometimes, it is necessary to suit the behaviour of the system to satisfy certain requirements. This explains the notion of *system level tailorability* that allows the hypermedia system designer to tailor the behaviour of the system itself.



System Level Tailorability

(The icon associated with this tailorability level is based on the puppet master metaphor (see [puppet master metaphor]).

System level tailorability aims to deliver services that affect the global behaviour of the system.

System level tailorability requires some knowledge about the internal architecture of the hypermedia system. Services attained through system level tailorability can be applied on different incarnations of the hypermedia framework: once we have implemented the technique in one framework incarnation, it requires little effort to reuse the code in other incarnations.

Typical examples of services that can be accomplished with system level tailorability concurrency control (i.e. manage concurrent access to shared data), logging (i.e. maintaining a log of navigation activities to provide backtracking features), caching (predict future behaviour on the basis of registered activities), authority control (check whether the user of the system has the privileges to see or modify information) and integrity control (control operations to preserve the consistency of the system's data structures). All these examples have in common that they change the behaviour of several modules in the system in a uniform way. *System level tailorability corresponds with wrapping uniform behaviour around different points residing on an axis of the Zypher design space.*

EXAMPLE

The Zypher hypermedia framework has been incarnated in a framework browser. Currently, the framework browser targets the VisualWorks\Smalltalk development environment documented using HTML and Microsoft Word. Porting the framework browser to a framework written in another programming language, developed using another development environment and documented with other documentation formats needs domain level tailorability. Such a port would require the construction of some specific classes and installing them in the framework. Writing such specific classes requires knowledge about the particular development environment and documentation applications involved.

On the other hand, a hypermedia system designer might decide to improve the general behaviour of the hypermedia system. An example may be to let the hypermedia system maintain a log of all navigation actions, which would allow the system to guide users backtracking the trail they have followed. Also, to improve the speed of the hypermedia system, one may want to manage a cache of recently accessed information fragments. Likewise, before opening or modifying a certain information fragment, one should be able to check whether the user is authorised to access that information. A final example would be to implement integrity control services to maintain the consistency of the hypermedia network in a distributed setting where multiple users share information.

Both levels of tailorability are important to suit the behaviour of the system to specific needs. However, since their aim is quite different, it is better to separate domain level tailorability from system level tailorability. Not only the separation results in a clearer design, but it also guides the software engineer when making the desired adjustments.

Problem

How can one make the domain level and system level functionality explicit, yet separate, in the design ?

Solution

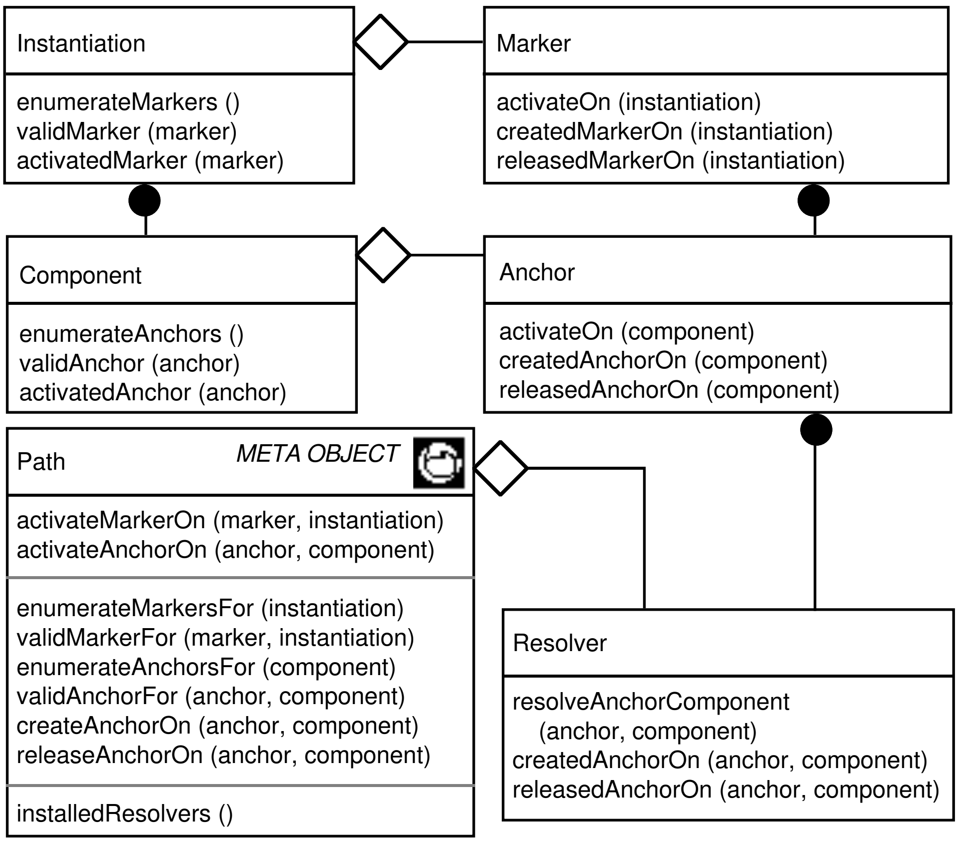
Domain level functionality is already furnished with the classes and interfaces specified in [resolver], [editor], [loader]. What is left implicit are the contracts between the different classes and that is precisely where the system behaviour is specified. So, to tailor the system behaviour one must represent those contracts explicitly.

Refer to the explicit contract representations as *meta-level objects*, which separates them from the participants in the contracts called *base level objects*. Rather than execute the contracts themselves, base level objects request the meta-objects to control the execution.

Have one meta-object for each axis in the design space, to attain a transparent design. Call *Path* the meta-object for the navigation layer, *Session* the meta-object for the presentation layer and *Hypertext* the meta-object for the storage layer.

Contract

a) Meta-object Protocol - Navigation



(See [class diagrams] for a short survey of the main elements in a class diagram)

Instantiation

All instantiations must delegate the `enumerateMarkers` and `validMarker` messages to the global Path object by means of the `enumerateMarkersFor` and `validMarkerFor` messages, but are allowed to perform additional operations. All instantiations implement an `activatedMarker` message sent by the global Path object when a navigation action on that particular instantiation was finished.

Marker

All markers must delegate the `activateOn` message to the global Path object by means of the `activateMarkerOn` message, but are allowed to perform additional operations. All markers implement the `createdMarkerOn`, `releasedMarkerOn` messages sent by the Path object to notify that the aggregation relation between an instantiation and a marker has been altered.

Component

All components must delegate the `enumerateAnchors` and `validAnchor` messages to the global Path object by means of the `enumerateAnchorsFor` and `validAnchorFor` messages, but are allowed to perform additional operations. All components implement an `activatedAnchor` message sent by the global Path object when a navigation action on that particular component was finished.

Anchor

All anchors must delegate the `activateOn` message to the global Path object by means of the `activateAnchorOn` message, but are allowed to perform additional operations. All anchors implement the `createdAnchorOn`, `releasedAnchorOn` messages sent by the Path object to notify that the aggregation relation between a component and an anchor has been altered.

Resolver

All resolvers implement a `resolveAnchorComponent` message sent by the global Path object to request for the determination of the target of a navigation operation. See [meta-meta-objects ~ contracts] and [navigation template ~ contracts] for further details.

All resolvers implement the `createdAnchorOn`, `releasedAnchorOn` messages sent by the Path object to notify that the aggregation relation between a component and an anchor has been altered.

Path

There is exactly one Path object for each hypermedia system.

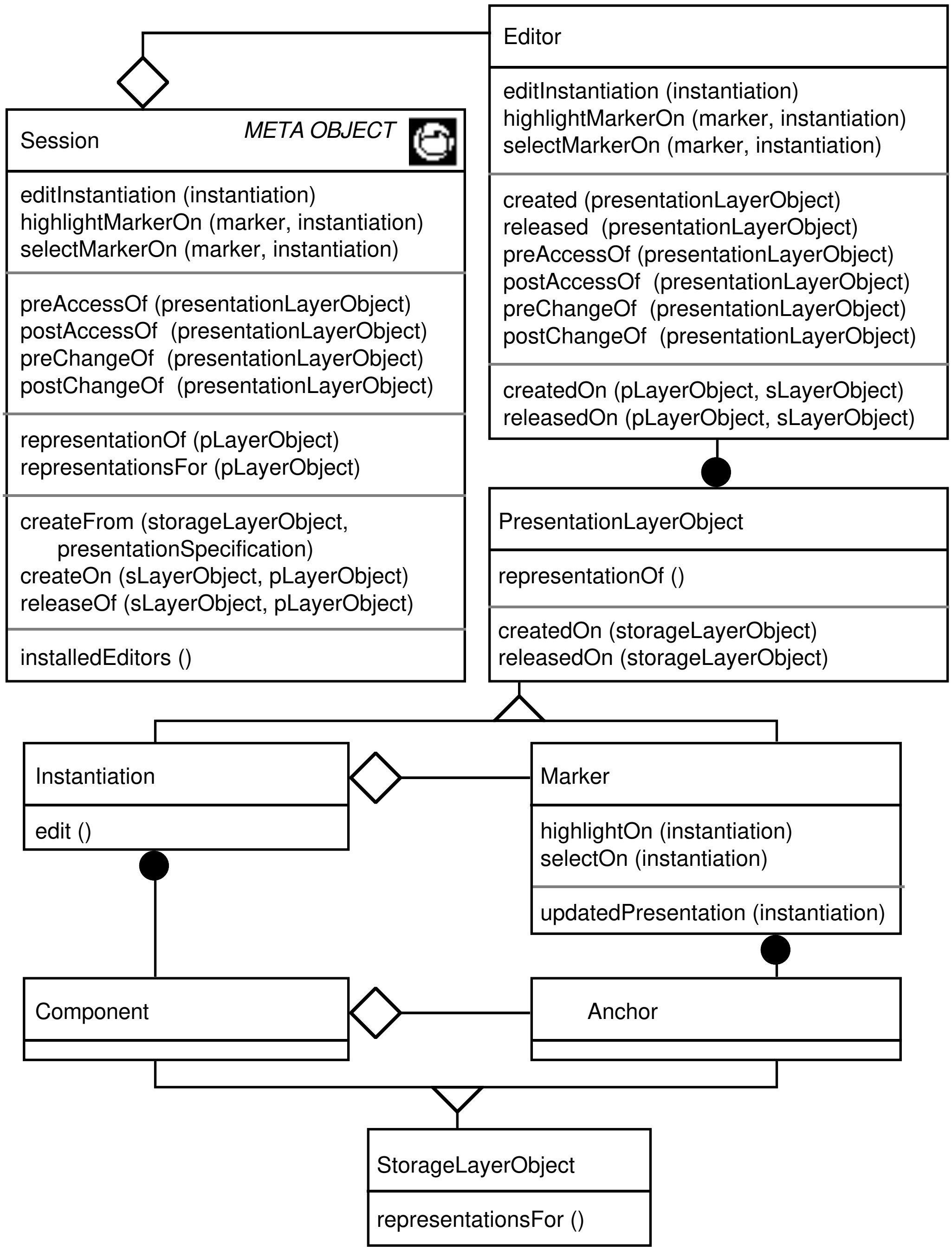
The Path implements an `activateMarkerOn` message as delegated by the marker object. This implementation must send an `activatedMarker` message to the instantiation associated with that marker; the implementation must finish with an `activateOn` message to the associated anchor (with the associated component as parameter).
The Path implements an `activateAnchorOn` message as delegated by the anchor object. This implementation must send an `activatedAnchor` message to the component associated with that anchor and an `resolveAnchorComponent` message to the associated resolver. See [meta-meta-objects ~ contracts] and [navigation template ~ contracts] for further details.

The Path is responsible for the maintenance of the aggregation relation between a component (instantiation) and an anchor (marker). For this purpose, the Path implements `enumerateMarkersFor`, `validMarkerFor`, `enumerateAnchorsFor` and `validAnchorFor` messages as delegated by the instantiation or component objects. The Path implements `createAnchorOn` and `releaseAnchorOn` message sent by any object that wants to alter the aggregation relation between an instantiation and a marker. The implementation must notify the anchor and resolver by means of the `createdAnchorOn`, `releasedAnchorOn` messages; the implementation must notify the marker by means of the `createdMarkerOn`, `releasedMarkerOn` messages.

Path / Resolver aggregation

For the navigation layer, the Path is an aggregation of resolvers. This should be interpreted that, at a certain moment, the Path contains a number of (zero or more) resolvers. The aggregation is allowed to vary over time. A Path implements the `installedResolvers` method to enumerate all the resolvers in the aggregation.

b) Meta-object Protocol - Presentation



(See [class diagrams] for a short survey of the main elements in a class diagram)

PresentationLayerObject

All markers and instantiations must delegate the `representationOf` message to the global Session object by means of the `representationOf` message, but are allowed to perform additional operations. All markers and instantiations implement the `createdOn`, `releasedOn` messages sent by the Session object to notify that the association between an instantiation (marker) and a component (anchor) has been altered.

Instantiation

All instantiations must delegate the `edit` message to the global Session object by means of the `editInstantiation` message, but are allowed to perform additional operations.

Marker

All markers must delegate the `highlightOn` and `selectOn` messages to the global Session object by means of the `highlightMarkerOn` and `selectMarkerOn` messages, but are allowed to perform additional operations. All markers implement an `updatedPresentation` message sent by the global Session object when an instantiation associated with that marker has been altered.

StorageLayerObject

All anchors and components must delegate the `representationsFor` message to the global Session object by means of the `representationsFor` messages, but are allowed to perform additional operations.

Editor

All editors implement an `editInstantiation` message sent by the global Session object to request for the presentation of an instantiation. All editors implement an `highlightMarkerOn` message sent by the global Session object to request the highlighting of a marker on an instantiation. All editors implement an `selectMarkerOn` message sent by the global Session object to request for the selection of a marker on an instantiation.

All editors implement the `created`, `released` messages sent by the Session object to notify that a presentation layer object has been created or removed. All editors implement the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` messages sent by the global Session object to notify that a presentation layer object is about to be, or has been, accessed or changed.

All editors implement the `createdOn`, `releasedOn` messages sent by the Session object to notify that the association between an instantiation (marker) and a component (anchor) has been altered.

Session

There is exactly one Session object for each hypermedia system.

The Session implements an `editInstantiation` message as delegated by the instantiation object; this implementation must send an `editInstantiation` message to the associated editor. The Session implements the `highlightMarkerOn` and `selectMarkerOn` messages as delegated by the marker object; these implementations must send an `highlightMarkerOn` or `selectMarkerOn` message to the associated editor.

The Session implements the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` messages as delegated by all presentation layer objects before or after any access or change. The implementations for the `preAccessOf`, `preChangeOf`, `postAccessOf` and `postChangeOf` messages must notify the associated editor by means of the corresponding messages.

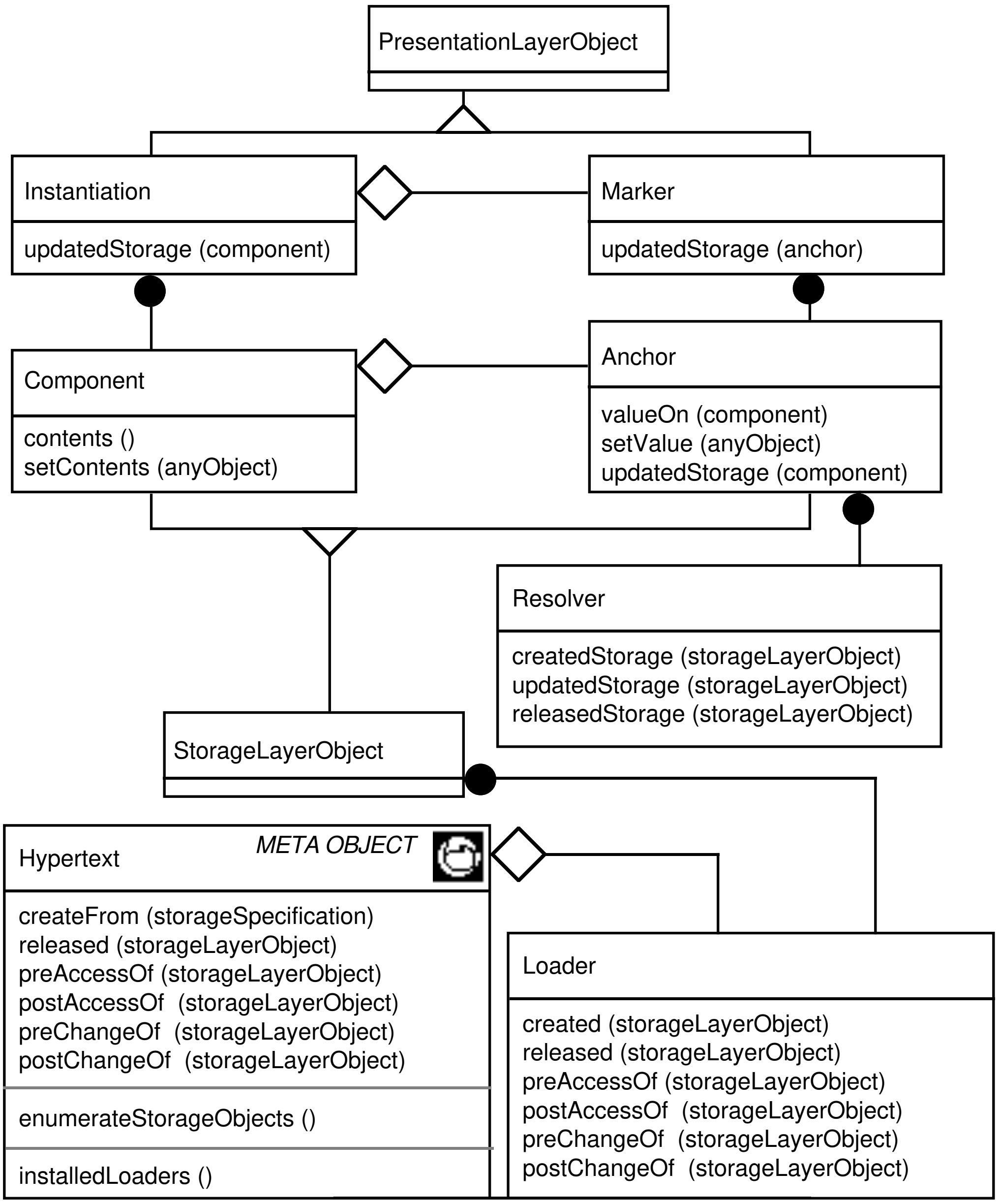
The Session is responsible for the maintenance of the 1-to-many association between a storage layer object (component or anchor) and a presentation layer object (instantiation or marker). For this purpose, the Session implements the `representationOf` and `representationsFor` messages as delegated by the instantiation or component objects. Also, the Session implements the `createOn` and `releaseOf` messages sent by any object that wants to alter the 1-to-many association between a component (anchor) and an instantiation (marker). The implementation must notify the instantiation (marker) and the editor by means of the `createdOn`, `releasedOn` messages.

The Session implements the `createFrom` message that creates a presentation layer object from a given storage layer object and a presentation specification. When the presentation object is created, it is associated with the storage layer object by means of the `createOn` message. See [meta-meta-objects ~ contracts].

Session / Editor aggregation

For the presentation layer, the Session is an aggregation of editors. This should be interpreted that, at a certain moment, the Session contains a number of (zero or more) editors. The aggregation is allowed to vary over time. A Session implements the `installedEditors` method to enumerate all the resolvers in the aggregation.

c) Meta-object Protocol - Storage



(See [class diagrams] for a short survey of the main elements in a class diagram)

Instantiation

All instantiations implement an `updatedStorage` message sent by the global Hypertext object when a component associated with that instantiation has been altered.

Marker

All markers implement an `updatedStorage` message sent by the global Hypertext object when an anchor associated with that marker has been altered.

Component

All components implement a `setContents` message sent by the associated loader when loading the contents of the component. All components implement a `contents` message sent by whatever object that needs the contents of the component; the returned value is based on the value set by the `setContents` message.

Anchor

All anchors implement a `setValue` message sent by the associated loader when loading the value of the anchor; this value is completely independent of the associated component. All anchors implement a `valueOn` message sent by whatever object that needs the value of the anchor on a certain component; the returned value is based on the value set by the `setValue` message.

All anchors implement an `updatedStorage` message sent by the global Hypertext object when a component associated with that anchor has been altered.

Resolver

All resolvers implement the `createdStorage`, `updatedStorage` and `releasedStorage` messages sent by the Hypertext object to notify that a storage layer object has been altered.

Loader

All loaders implement the `created`, `released` messages sent by the Hypertext object to notify that a storage layer object has been created or removed. All loaders implement the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` messages sent by the global Hypertext object to notify that a storage layer object is about to be, or has been, accessed or changed.

Hypertext

There is exactly one Hypertext object for each hypermedia system.

The Hypertext implements the `preAccessOf`, `preChangeOf`, `postAccessOf`, `postChangeOf` messages as delegated by all storage layer objects before or after any access or change. The implementations for the `preAccessOf`, `preChangeOf`, `postAccessOf` and `postChangeOf` messages must notify the associated loader by means of the corresponding messages. All resolvers must be notified after changes by means of the `updatedStorage` message.

The Hypertext implements the `createFrom` message that creates a storage layer object from a given storage specification. The Hypertext implements a `release` message sent by whatever object that wants to remove a storage layer object. The implementation must notify the associated loader by means of the `created` and `released` messages. All resolvers must be notified by means of the `createdStorage` and `releasedStorage` messages. See [meta-meta-objects ~ contracts] and [navigation ~ contracts] for further details.

The Hypertext implements the `enumerateStorageObjects` message to enumerate all existing component or anchors.

Hypertext / Loader aggregation

For the storage layer, the Hypertext is an aggregation of loaders. This should be interpreted that, at a certain moment, the Hypertext contains a number of (zero or more) loaders. The aggregation is allowed to vary over time. A Session implements the `installedLoaders` method to enumerate all the resolvers in the aggregation.

Motivation

The motivation to introduce meta-objects (with a corresponding meta-object protocol) into the design of the system was to attain system level tailorability. We chose as meta-objects the explicit representation of the contracts between the base level objects. Why does this choice allow us to provide system level tailorability ?

a) Behaviour

Typical examples of services that must be accomplished with system level tailorability are things like logging (maintaining a log of certain activities to provide backtracking features), authority control (check whether the user of the system has the privileges to do certain operations) and caching (predict future behaviour on the basis of registered activities). Such functionality is primarily concerned with the *behaviour* of the system, which is in some sense orthogonal to the base level functionality. To tailor the system behaviour one wants to control all occurrences of a particular message that is being sent, regardless of the objects involved.

The messages that matter in the design are specified in the static part of the contracts (i.e. the interface of the different classes), thus part of the design. However, the dynamic part of the contracts (i.e. the decision when a certain message is sent) is delegated to the implementation and it is precisely the dynamic part of the contracts that determines the behaviour. So, if the dynamic part of the contracts is delegated to the implementation, one must verify all classes that may send or receive an important message to control all occurrences of that message. On the contrary, if we create meta-objects that control the dynamic parts of the contracts, all occurrences of important messages are funnelled through a single meta-object, independent of the participants involved. To control the system behaviour, only the single funnelling point in the meta-object must be observed.

EXAMPLE

To implement a backtrack function on the trail of all activated navigation sources one must log all occurrences of the `activateOn` message sent to markers. Without the Path meta-object, this means that one must adjust all the implementation of all `activateOn` messages in all the marker classes, or that one must adjust the implementation of all editor classes that sent the `activateOn` message. With the Path meta-object, it suffices to adjust the implementation of the `activateMarkerOn` message in the sole Path meta-object that exists in the hypermedia system.

To incorporate authority control without the Session meta-object, one must adjust all implementations of all `edit` messages in all instantiation classes or adjust methods that send the `edit` message. With the presence of the Session meta-object, one can do with an adjustment of the `editInstantiation` method in the sole Session meta-object.

A possible technique to speed up the response time of the system, is to maintain a cache of all recently accessed components. Without the Hypertext meta-object, if one wants to detect what components have been accessed recently, one must adjust all implementations of all `contents` messages in all component classes or adjust all implementations of `preAccessOf` (or `postAccessOf`) messages in all loader classes. With the presence of the Hypertext meta-object, it is sufficient to adjust the implementation of the `preAccessOf` (or `postAccessOf`) message in the sole Hypertext meta-object that exists in the hypermedia system.

b) Structure

Contracts include, besides a static (i.e. the messages understood by a given class) and a dynamic part (i.e. the decision when a certain message is sent) also a structural part (i.e. establish associations between objects). Along with making the dynamic part of the contracts explicit, the meta-objects do control the structural parts. This suggests that besides the behaviour level, there exists system tailorability on the *structural* level as well. Indeed, services like integrity control depend largely on the possibility to verify all operations modifying the structure. Again, if all such operations are funnelled through a single meta-object, it is much easier to verify than if such operations are spread over all the classes of the system.

EXAMPLE

In a shared and distributed hypermedia system, one needs a concurrency control mechanism to maintain the consistency of single information fragments. Without the Hypertext meta-object, this implies that all implementations of all `contents`, `setContents`, `valueOn` and `setValue` messages (or all the methods sending these messages) must be adapted. With the Hypertext meta-object, it suffices to adjust the implementation of the `created`, `released`, `preAccessOf`, `postAccessOf`, `preChangeOf` and `postChangeOf` messages in the sole Hypertext meta-object that exists in the hypermedia system.

To maintain the consistency of the navigation network, one should be able to adapt the system's data structures when a component or an anchor is deleted. Without the Hypertext meta-object, one must adapt all implementations of all `released` messages on all loaders. With the Hypertext meta-object, one can do with a modification on the sole Hypertext meta-object that exists in the hypermedia system.

c) Multiple Inheritance

Often, people point out that multiple inheritance may provide solutions for the problems sketched above. This is not entirely correct, as multiple inheritance should be considered an implementation issue and not a design issue.

It is true that if we want to adapt all occurrences of a particular message, we can create a so-called mixin class that includes that message in its interface and provides the necessary extra functionality in the implementation. However, this mixin class must be included in the multiple inheritance hierarchy and then there are two cases to consider. The first case a) excludes the possible existence of a mixin class from the design. This implies that we are obliged to adapt all the classes of the system that do implement this message: for each of these classes, we must include the mixin class in the super class chain. Moreover, the exclusion of the mixin classes from the design, implies that subclasses are not forced to forward the message to their super, so for each of the classes we must check whether the implementation of the message forwards to the super. The second case b) includes the existence of such mixin class in the design of the framework. However, such a mixin class is then a meta-object since it made a part of the contract explicit. Case a) leads to the conclusion that if the possibility of multiple inheritance is not included in the design there are no benefits to expect, as software engineers still are forced to adapt all existing classes. Case b) implies that including multiple inheritance in the design is equivalent to the definition of meta-objects.

Issues

Naming

Meta-objects are abstract: they do not have a concrete representation in the problem domain for which the system was designed. As a consequence, choosing names for meta-objects is hard. One option is to use the name 'X meta-object', where X is the name of an aspect in the base level. This quickly leads to very long names unworkable in practice (i.e. navigation layer meta-object, presentation layer meta-object, storage layer meta-object). We have chosen the other option and have chosen short names, based on existing data models in the hypermedia research field. The Dexter model [Halasz,Schwartz'90], furnished the 'Hypertext' and 'Session' names because these objects are responsible for the management of the storage and presentation layer respectively. The name 'Path', stems from the work of Zellweger [Zellweger'89].

Why is this 'Meta' ?

There is an important question that still needs an answer: why is this called 'meta' ? What makes this different from usual software design ? What is so special about this abstraction ?

The term meta is generally connoted with the notion of *reflection*, i.e. the ability of a system to inspect and modify representations of its own activities. Reflection is an intriguing idea —certainly within computer science— but is mostly considered an academic issue. Reflection has been studied in the area of artificial intelligence and the design of computer languages for quite a long time now (i.e. [Maes'87], [Kiczales,Rivières,Bobrow'91], [Steyaert'94]). There, it has been shown that reflection eases extensibility (i.e. define a small and fixed kernel language and use that kernel to extend the language expressiveness), compatibility (i.e. backward compatibility with older definitions of the language) and efficiency (i.e. differ the implementation strategy to optimise behaviour). Moreover, since a reflective system is able to monitor its own activities, powerful tools like debuggers and code optimisers can be constructed more comfortably.

Recently the idea has been applied on the design of systems other than programming languages (i.e. [Rao'91]), leading to what has been called *implementational reflection* (or sometimes *open implementations*). A system with implementational reflection is able to inspect and/or change the implementational structures of its subsystems. Implementational reflection does not directly provide solutions for the problem domain the system has been designed for, but it does contribute to the internal organisation and the external interface of that system. This suggests that what we have been calling system level tailorability is indeed a feature that can be attained with implementational reflection.

To explain why the funnel objects make the hypermedia system a reflective one, we turn to the definitions found in [Maes'87]. There, *a reflective system is defined as a system which is about itself in a causally connected way*. We elaborate on the three main ideas in this definition (i.e. system, about-ness and causal connection) to make things more precise. A '*system*' is software running on a computer with the intention to answer questions about and/or support actions in some domain. A system incorporates internal structures representing its domain, that is why a system is said to be '*about*' its domain. A system is said to be '*causally connected*' to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect on the other. In an object-oriented implementation of a system, the parts of the system that represent causally connected internal structures belong to the meta-level, hence are called meta-objects.

The definition of causal connection implies that a causally connected system may cause changes in the problem domain by a mere change in the internal representation of that problem domain. As a consequence (since a reflective system incorporates structures that are causally connected to itself) a reflective system can modify itself by changing its internal representation.

To argue why the funnel objects (i.e. Path, Session, Hypertext; see the [contracts] section) defined in the previous sections are meta-objects, we must prove that these objects are (a) about the hypermedia system in (b) a causally connected way. The proof follows from the insight that the funnel objects are *explicit representations of the contracts defined between the objects on the base level*. Indeed, the important messages are specified in the static part of the contracts (i.e. the interface specified in [resolver], [editor], [loader]), thus part of the design. However, without the funnel objects, the dynamic part of the contracts (i.e. the decision when a certain message is sent) is delegated to the implementation and it is precisely the dynamic part of the contracts that determines the system's behaviour. If the design is extended with the specially created Path, Session and Hypertext objects, the dynamic parts of the contracts are explicitly available, since all occurrences of all important messages arrive at, or originate from such funnel objects. Knowing that the specially created Path, Session and Hypertext objects are representations of the dynamic parts of the contracts between the base level objects —specifying how the system should behave under certain conditions— , they are by definition 'about' the system. Moreover, they are an explicit representation of the contracts, which makes them 'causally connected' to the system. Changing the implementation of a funnel object has immediate effect on the subsequent behaviour of the system. And since the only legal way to change a framework's design is to change its contracts, we can be sure that changing the implementation of the funnel objects is the only way to change the general behaviour of the system.

The Meta-Linguistic Abstraction View

When dealing with meta-levels and reflection, it is important to note that there are other possibilities to look at reflection than the system view we applied in the above argumentation. Probably, the most popular one is the meta-linguistic abstraction view, among others expressed in [Abelson,Sussman'84]. The meta-linguistic abstraction view starts from the assumption that, for all computer programs, there exists some kind of a descriptive *language* used to express problems on some level of abstraction. An *evaluator* is then a computer program that —when applied to an expression of that language— performs the actions required to realise the meaning of that expression; such an evaluator is called a meta-system. If the evaluator is tailorable by the software engineer using the descriptive language, then it is possible to control the actions realising the expression and we have already seen that this eases the implementation of many useful system services.

The meta-linguistic abstraction view works very well in the domain of programming languages, because there the notions of 'language' and 'evaluator' have a concrete counterpart in the problem domain. Although it is not hard to regard any program as the evaluator of some language, it is much harder to draw a clear line between the 'language' and the 'evaluator' in typical software systems. Certainly, if one looks at the internal architecture of an object-oriented software system, it is almost impossible to define objective criteria that help in judging whether a given object is part of the base level or the meta-level. This is largely because the meta-linguistic abstraction view requires that one must identify the language before it is possible to distinguish the evaluator. In almost all problem domains, there is no well specified language available, or —even worse— there are several degrees of languages possible. We prefer the system view on meta-systems, because the definition of a meta-system is based on 'about-ness' and 'causal connection', properties much easier to recognise in typical application domains than 'language' and 'evaluator'.

EXAMPLE

> In the problem domain of open hypermedia systems, several languages (in the meta-linguistic sense of the word) manifest themselves. There are command languages to instruct viewer applications and repositories to perform the necessary actions; the algorithms that determine the navigation relations are specified in some language as well. So the editor, loader and resolver objects in the Zypher frameworks can be seen as meta-objects interpreting languages formed by component-anchor-instantiation-marker structures. On the other hand, when an editor has detected a mouse click on a sensitive region, it requests the marker to interpret the event and perform the necessary

actions. So the marker can be seen as an interpreter for the editor-marker-instantiation structure. Also, loaders may detect that some information has been modified in the repository and request a component and the associated anchors to update their contents: then the component and anchor play the role of interpreters for the loader object. The same applies for resolvers discovering disabled navigation endpoints: they may request the anchors to disable themselves, in which case the anchor objects interpret resolver messages.

Consequences

There is an important question left unanswered here, and that is why there should only be one instance for each Path, Session and Hypertext object ? We postpone the answer to this question until [navigation template].

Relations

Where To Go Next ?

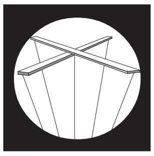
∇ People reading the Zypher design pattern documentation for the first time should have learned about the meta-object protocol in the Zypher framework and how this allows for system level tailorability. The obvious next step is then the [meta-meta-objects] pattern, that provides an interface for configure the modules of the hypermedia system.

Other Catalogues

The Path, Session and Hypertext meta-objects have much in common with the Strategy pattern as defined in [GammaEtAl'93], p.315-323. The strategy pattern is designed to encapsulate each member of a family of algorithms, and makes them interchangeable for the clients that use them. From this perspective, the resolver, editor and loader objects can be seen as three different families of algorithms, while the anchor, component, instantiation and marker objects are the clients that use them. The Path, Session and Hypertext meta-objects then participate in the strategy pattern playing the role of the context, in the sense that they dispatch the client calls to the appropriate strategy object. As a consequence, the meta-objects offer the same degree of interchangeability as does the strategy pattern. However, the role of the meta-objects is slightly different from the role the context plays in the design pattern. Meta-objects are not supposed to provide another (higher level) interface for the algorithms defined in the strategy objects, as is implicitly assumed by the strategy pattern. In fact, the interfaces of the meta-objects and the base level objects should overlap as much as possible, as it is the primary role of the meta-object to provide a funnelling point, while the primary role of the context object in the strategy pattern is to dispatch client calls. Note that the strategy pattern is vague about the way the context object chooses the appropriate strategy. For meta-objects this decision is controlled by the meta-meta-object.

Meta-meta-objects: Configuration Level Tailorability

Intent



Provide an interface to configure the modules of the hypermedia system.

Analysis

Zypher is a modular hypermedia framework. To construct a hypermedia system, the system designer assembles the necessary base level objects and controls their behaviour using the meta-level objects (see [meta-objects]). The base level objects are instances of classes installed in the system.

The modularity property is best illustrated with the Zypher design space [figure 22], which consists of three orthogonal design axes. The *storage* axis enumerates all possible repositories that may be incorporated; the *presentation* axis enumerates all possible viewer applications that can be applied to manipulate a particular information fragment and the *navigation* axis enumerates all algorithms that determine the navigation relationships. A hypermedia system constructed with Zypher is the sum of the classes installed in the different axes; so a hypermedia system covers a three-dimensional cube within the Zypher design space[10].

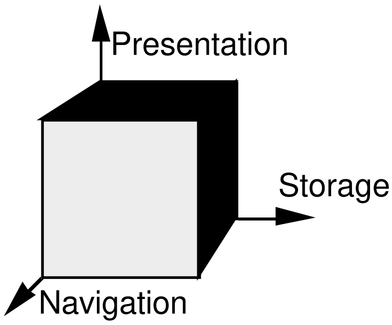


Figure 22: Zypher design space

A particular hypermedia system is created by a hypermedia system designer through the installation of classes. At run-time those classes produce objects that are assembled in various configurations to exhibit the desired behaviour. To achieve a maximum of flexibility, the configuration of the objects should not depend on the implementation of the classes. Then the system's configuration is able to vary according to the system's context, the classes installed, the preference settings, … .

EXAMPLE

> An example that illustrates the configurability principle is the interpretation of the URL (universal resource locator) format for anchors as defined in the HTML-specification (see [Berners-LeeEtAl'94]). HTML-documents embed the addresses of navigation targets in their documents using the URL format and the same technique may be applied for the Microsoft Word documents containing design pattern documentation.

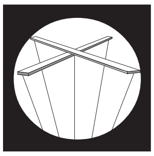
---

10 Actually, the arrangement of the Zypher design space (see [figure @@]) is a little misleading. The three axes of the design space are not perfectly orthogonal: some of the modules installed in one axis will not support all of the modules in all other axes. This means that a hypermedia system constructed with Zypher is not a Cartesian product but a relation —in the mathematical sense of the word— between modules installed in the different orthogonal axes.

The URL format is open in the sense that it is prefixed by a keyword identifying the target address space, followed by the actual address in a format depending on the keyword prefix. So, the list of interpretable keywords delimits the linking potential of the hypermedia system. A configurable hypermedia system allows to extend the list of interpretable keywords without affecting the implementation of the classes dealing with the resolution, loading and presentation of information stored in a particular URL address space.

Another example is about the system's configuration across platforms. Framework documentation is organised as a collection of design patterns stored in files and displayed with different viewer application (i.e. a HTML-browser and the Microsoft Word applications). This situation has two important consequences: some viewer applications are not be available on all platforms (i.e. the Microsoft Word application is not available on UNIX platforms) and some viewer applications are preferable over others (i.e. usually the Microsoft Word viewer application is preferred over the HTML-browser, although some users may express other preferences). So, the determination of the optimum configuration, must be based on the underlying hard- and software platform and the state of particular user preferences. A configurable hypermedia system allows system designers to influence this determination process, without modifying the implementation of the participating classes.

To provide such configurability, the hypermedia system designer should be able to influence the system decision on which class to use in what occasion. This implies a third level of tailorability besides the previously discussed domain tailorability and system tailorability (see [meta-objects]).



Configuration Level

(The icon associated with this tailorability level is based on the puppet master metaphor (see [puppet master metaphor]).

Configuration level tailorability aims to provide a 'plug and play' system, where the coordination between the system modules is adapted without changing their internal implementation.

Configuration level tailorability requires a deep knowledge about the internal architecture of the hypermedia system; however technical details about individual modules do not matter.

Typical usage of configuration level tailorability is to make the system run-time extensible. For instance, the table of helper applications in most world-wide web browsers, where users can associate a viewer application with a given document type is about the configuration between the storage and presentation axis. The Java approach [Java'95] is another example, where world-wide web browsers load system modules for handling unknown document formats. The final example is protocol negotiation, where system modules communicate to choose the optimal protocol for exchanging data; protocol negotiation is valuable in cross-platform communication. *Configuration level tailorability corresponds with changing the relationship (in the mathematical sense of the word) between the points on the axes in the Zypher design space.*

The three levels of tailorability are important to suit the behaviour of the system to particular needs. However, to attain a clear design, it is better to provide a different interface for all levels of tailorability.

Problem

How can one make the notion of configuration level tailorability explicit —yet separate— in the design ?

Solution

Elsewhere (see [meta-objects]) we have shown that system level tailorability is achieved by the introduction of meta-level objects, i.e. explicit representations of the contracts between base level objects. Those meta-objects are defined as objects controlling all operations concerning a particular axis in the Zypher design space (i.e. Path for the navigation axis, Session for the presentation axis and Hypertext for the storage axis). As we can expect from this definition, there are more operations defined on meta-objects, as the ones that follow from funnelling base level operations. A quick look at the design with the meta-level objects (see [meta-objects ~ structure], [meta-objects ~ contracts]) learns that the introduction of the meta-objects adds operations for the maintenance of the aggregation relationships Path-Resolver, Session-Editor and Hypertext-Loader (i.e. `installedResolvers`, `installedEditors`, `installedLoaders`). The role of these aggregation relationships is to specify what kind of resolvers, loaders and editors are installed in the hypermedia system, which corresponds to the management of the available peripheral systems.

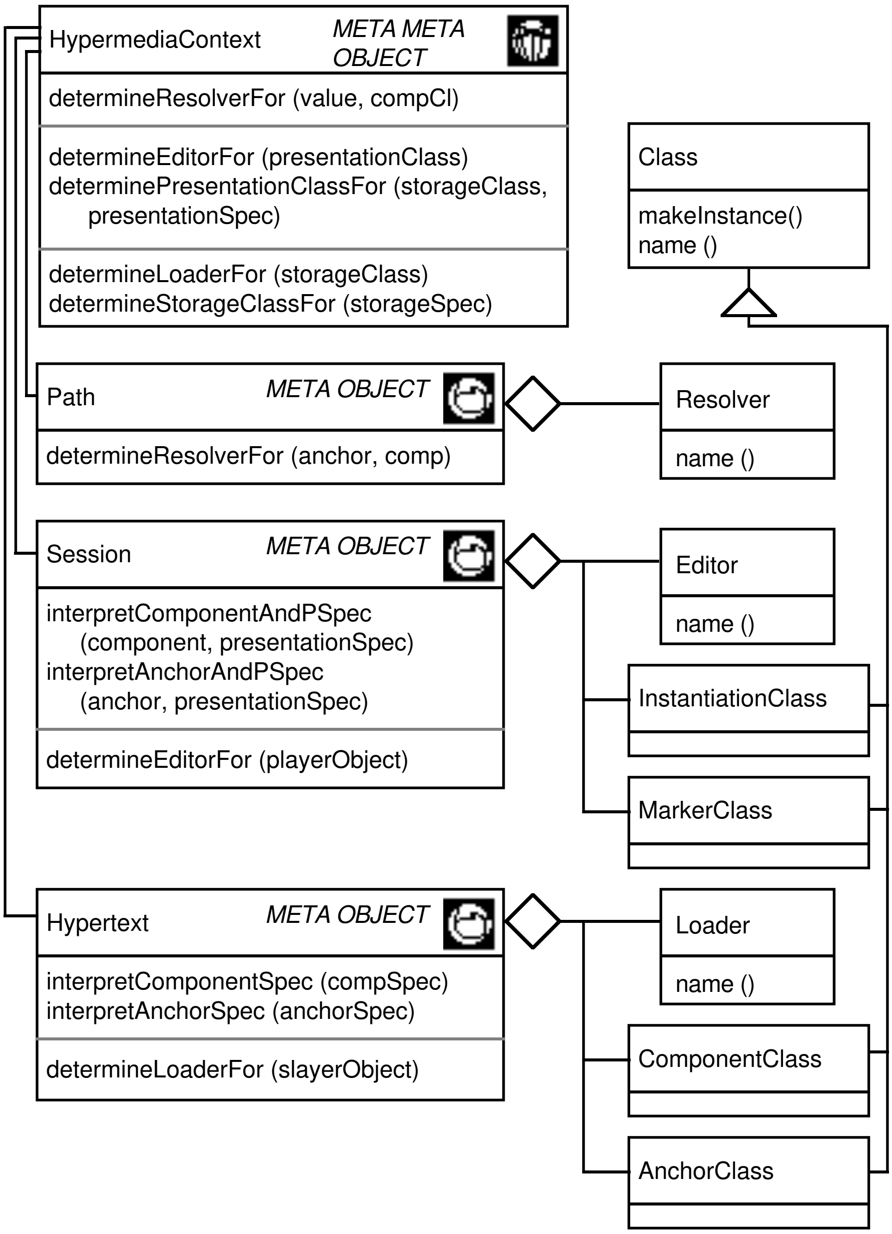
Also, the contracts on the meta-objects include operations to create, query and release associations between objects (StorageLayerObject - PresentationLayerObject in Session; Instantiation - Marker and Component - Anchor in Path) plus relations between objects and peripheral objects (PresentationLayerObject - Editor in Session; StorageLayerObject - Loader in Hypertext; Anchor - Resolver in Path). The role of these operations is to govern the connections between the internal elements of the hypermedia system and to control the links with the outside world.

Moreover, if the meta-objects must control all operations concerning a particular layer, then the meta-objects must supervise the creation of potential objects constituting the running hypermedia system. This implies that the contracts for those meta-objects must include aggregation relationships specifying the available classes that create the actual objects (i.e. Hypertext - ComponentClass, Hypertext - AnchorClass, Session - InstantiationClass, Session - MarkerClass). These aggregation relationships manage the classes that produce the objects that establish the actual system and are depicted in [meta-meta-objects ~ contracts].

The previous paragraphs give short descriptions of the contracts introduced by the meta-level objects (see also [meta-objects ~ structure] and [meta-objects ~ contracts]) showing that —besides funnelling the navigation, presentation and storage layer operations— the meta-objects do implement the configuration of the hypermedia system. However, the implementation of these configuration messages is not made explicit in the design, which suggests that an explicit representation of the contracts defined on meta-level objects leads to the required configuration level tailorability.

Call the explicit representation of contracts between meta-objects a *meta-meta-object*. Have a single meta-meta-object for each hypermedia system and call it the HypermediaContext.

Contract



(See [class diagrams] for a short survey of the main elements in a class diagram)

Path

The Path contains a number of resolvers accessible by name. The Path is installed in the global HypermediaContext.

Each Path implements a `determineResolverFor` message (accepting an anchor and a component as parameter). Each Path must use this message in the implementation of the `activateAnchorOn` message to determine the resolver to apply. The implementation of the `determineResolverFor` message must forward the message (with the value of the anchor and the class name of the component as parameters) to the global HypermediaContext object to request for the name of the resolver.

Session

The Session contains a number of editors, instantiation classes and marker classes accessible by name. The Session is installed in the global HypermediaContext.

Each Session implements an `interpretComponentAndPSpec` (accepting a component and a presentation specifier as parameters) and an `interpretAnchorAndPSpec` (accepting an anchor and a presentation specifier as parameters) used to create an instantiation or marker object. Each Session must use this message in the implementation of the `createFrom` message to create the actual object. The implementation of the `interpretComponentAndPSpec` and `interpretAnchorAndPSpec` messages must send a `determinePresentationClassFor` (with the class name of the storage layer object and the presentation specifier as parameter) to the global HypermediaContext object to request for the name of the class.

Each Session implements a `determineEditorFor` message (accepting a presentation layer object as parameter). Each Session must use this message in the implementation of the `editInstantiation`, `highlightMarkerOn`, `selectMarkerOn`, `preAccessOf`, `postAccessOf`, `preChangeOf`, `postChangeOf` messages to determine the editor to apply. The implementation of the `determineEditorFor` message must forward the message (with the class name of the presentation layer object as parameter) to the global HypermediaContext object to request for the name of the editor.

Hypertext

The Hypertext contains a number of loaders, component classes and anchor classes accessible by name. A Hypertext is installed in the global HypermediaContext.

Each Hypertext implements an `interpretComponentSpec` (accepting a component specifier as parameter) and an `interpretAnchorSpec` (accepting an anchor specifier as parameter) used to create a component or anchor object. Each Session must use this message in the implementation of the `createFrom` message to create the actual object. The implementation of the `interpretComponentSpec` and the `interpretAnchorSpec` messages must send a `determineStorageClassFor` message (with the component specifier or anchor specifier as parameter) to the global HypermediaContext object to request for the name of the class.

Each Hypertext implements a `determineLoaderFor` message (accepting a storage layer object as parameter). Each Hypertext must use this message in the implementation of the `preAccessOf`, `postAccessOf`, `preChangeOf`, `postChangeOf` messages to determine the loader to apply. The implementation of the `determineLoaderFor` message must forward the message (with the class name of the storage layer object as parameter) to the global HypermediaContext object to request for the name of the loader.

HypermediaContext

There is exactly one HypermediaContext object for each hypermedia system. A HypermediaContext is associated with one Path, one Resolver and one Hypertext.

The HypermediaContext implements a `determineResolverFor` (accepting an anchor value and the name of a component class as parameter) message as delegated by the Path object; the implementation returns the name of an installed and active resolver.

The HypermediaContext implements a `determineEditorFor` (accepting the name of an instantiation class or marker class as parameter) message as delegated by the Session object; the implementation returns the name of an installed and active editor.

The HypermediaContext implements a `determinePresentationClassFor` (accepting the name of a component class or anchor class and a presentation specifier as parameter) message as delegated by the Session object; the implementation returns the name of an instantiation or marker class.

The HypermediaContext implements a `determineLoaderFor` (accepting the name of a component class or anchor class as parameter) message as delegated by the Hypertext object; the implementation returns the name of an installed and active loader. The HypermediaContext implements a `determineStorageClassFor` (accepting component or anchor specifier as parameter) message as delegated by the Hypertext object; the implementation returns the name of a component or anchor class.

Class (InstantiationClass, MarkerClass, ComponentClass, AnchorClass)

All classes implement a `makeInstance` message that returns a new instance of that class. All classes implement a `name` message that returns a name that uniquely identifies the class within the system.

Resolver, Editor, Loader

All resolvers, editors and loaders implement a `name` message that returns a name that uniquely identifies the resolver, editor, loader within the system.

Motivation

The motivation to introduce a meta-meta-object (with a corresponding meta-meta-object protocol) into the design of the system was to attain configuration level tailorability. We chose as meta-meta-object the explicit representation of the contract defined on the meta-objects. Why does this choice allow us to provide configuration level tailorability ?

a) Configuring Behavioural Objects

Within the base level objects defined elsewhere one may recognise two categories of objects. Some of them are intended to provide building blocks for structuring aspects of information (i.e. Component, Instantiation, Anchor and Marker); others intend to encapsulate algorithms, behaviour and commands sets (i.e. Resolver, Editor, Loader). These categories stem from decoupling structure and behaviour, which eases configurability and extensibility (see [resolver ~ motivation], [editor ~ motivation], [loader ~ motivation]). Decoupling structure from behaviour is known to be good practice in software design, so it is a recurring theme in almost all frameworks.

When the design of a system decouples behaviour from structure, this implies that the system's implementation must decide how and when the two must be joined. The processes that implement this decision are crucial in the system's configuration, because attaching behaviour to structure actually specifies how the system behaves under certain conditions.

Reviewing the design (i.e. [meta-objects ~ contract]) with this knowledge in mind, reveals that there is one 'behavioural' object for each layer (i.e. Resolver for the Navigation Layer, Editor for the Presentation Layer and Loader for the Storage layer). When funnelling base level operations, the meta-objects (i.e. Path, Session and Hypertext) dispatch the structure operations to the appropriate behavioural equivalent, thus the meta-objects implement at least part of the system's configuration.

In design of the meta-level (see [meta-objects ~ contract]), we know what messages implement the attachment of a behavioural object to a structure object. However, the decision itself is not explicitly present in the design, which gets rectified in the meta-meta-layer (see [meta-meta-objects ~ contract]) with the introduction of the `determineResolverFor`, `determineEditorFor`, `determineLoaderFor` messages that consult the meta-meta-object to determine the name of the appropriate behavioural object.

To conclude, the meta-meta-object is responsible for configuration level tailorability, because it provides an explicit decision on what behavioural object interprets a given combination of structure objects.

EXAMPLE

> A good example of behavioural configuration is the interpretation of a keyword identifying an URL address space. As described in the analysis section (see [meta-meta-objects ~ analysis]) a configurable hypermedia system allows to extend the list of interpretable keywords without affecting the implementation of the classes dealing with the resolution, loading and presentation of information stored in a particular URL address space. In the Zypher framework, the crucial decision is the mapping of the keyword on the resolver that interprets the address. When the Path object is asked to activate a given anchor on a given component (by means of the `activateAnchorOn` message), the Path object must dispatch this message to the appropriate resolver. To request for the name of the resolver, the Path sends the `determineResolverFor` message (with the value of the anchor as parameter) to the HypermediaContext object. This HypermediaContext object fetches the prefix keyword from the anchor value and queries a look-up table to find a priority list of candidate resolvers. This priority list is matched with the list of active resolvers to find the name of the active resolver with the highest priority.

b) Configuring Structure

Having identified the difference between structural objects (i.e. Component, Instantiation, Anchor and Marker) and behavioural objects (i.e. Resolver, Editor, Loader) we have seen that the meta-meta-object is explicitly consulted when a behavioural object is associated with a structural object. However, associating structure with behaviour is but one aspect of a system's configuration — another important aspect is what structure should be created ? What classes should be instantiated to create a structure that models to the situation in the application domain ?

Just like in the behaviour/structure case, the design of the meta-level (see [meta-objects ~ contract]) includes messages that implement the decision which class to instantiate. The decision itself is made explicit in the meta-meta-layer (see [meta-meta-objects ~ contract]) with the introduction of the `interpretComponentAndPSpec`, `interpretAnchorAndPSpec`, `interpretComponentSpec`, `interpretAnchorSpec` (see [meta-objects ~ contract]), messages that consult the meta-meta-object to determine the name of the appropriate class.

So the second argument sustaining the viewpoint that the meta-meta-object implements configuration level tailorability, is that it provides an explicit decision on what structure objects are created to model the situation in the application domain.

EXAMPLE

> An example of structural configuration is the decision on the component-instantiation pair representing a particular design pattern document. A component representing a design pattern document may be viewed by a Microsoft Word application or a HTML; this should be viewed by

Issues

Why Meta-Meta ?

The HypermediaContext object belongs to a meta-level as it is an explicit representation of a framework contract. However, since that framework contract is defined on the meta-level, the HypermediaContext object is part of the meta-level of the meta-level of the framework, hence the name meta-meta-object.

Consequences

There is an important question left unanswered here, and that is why there should only be one instance of the central HypermediaContext object ? We postpone the answer to this question until [navigation template].

Relations

Where To Go Next ?

∇ People reading the Zypher design pattern documentation for the first time should have learned about the meta-object protocol in the Zypher framework and how this allows for system level tailorability. The obvious next step is then the [meta-meta-objects] pattern, that provides an interface for configure the modules of the hypermedia system.

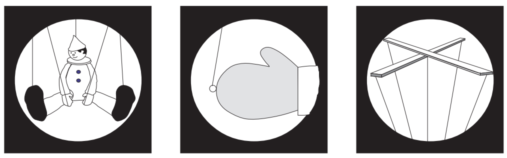
Other Catalogues

The meta-level objects play the role of object factories in object-oriented frameworks. The Session and Hypertext objects are abstract factories[GammaEtAl'93], p.87-95.

CHAPTER *8*

Protocols in an Open Hypermedia Framework

Navigation Template: Specifying The Control Flow



Intent

Provide a high level specification for the control flow of the hypermedia system to make clear what parts must be extended to achieve a certain effect. This specification is independent of particular information repositories, viewer applications or navigation resolution algorithms.

Analysis

An object-oriented framework is a set of co-operating classes that makes up a reusable design for a given problem domain. This is in some sense opposed to a software library, which is a set of routines where the code may be reused in different implementations. It is widely recognised that object-oriented frameworks aim at design reuse, while software libraries aim at code reuse.

One of the important differences between software libraries and frameworks is the so-called *Hollywood principle* ("don't call us, we'll call you") [Cotter,Potel'95]. This principle is the basis to understand how an object-oriented framework promotes design reuse and has to do with the definition of the control flow. With a software library approach, a software engineer writes code that calls the routines supplied by the library, so it is the software engineer that is defining the control flow. With the framework approach, a software engineer writes code that is installed in the framework and the framework calls that code.

The Hollywood principle states that a framework manages the control flow, which implies that the definition of the control flow is an essential ingredient of a framework's design specification. So, an important question to answer is how much of the control flow is incorporated in the design.

So far ([interoperability], [navigation], [resolver], [editor], [loader], [events], [meta-objects], [meta-meta-objects]), we have established design specifications for domain level, system level and configuration level tailorability. The emphasis of these design specifications lay on the definition of the elements constituting the framework structure and the role those elements play in that structure (i.e. the "Contract" section). As far as control flow concerns, this is too detailed to be useful. As a result, the control flow of the overall system is unclear, which makes it hard to understand what part of the system must be extended to achieve a certain effect.

Note that —according to the Hollywood principle— most of the functionality of the system is defined in the classes supplied by the software engineer tailoring the framework for a particular application. To attain a clear specification of the overall control flow, we must ensure that the specification is independent of particular information repositories, viewer applications or navigation resolution algorithms.

Problem

According to the Hollywood principle, provide a specification for the control flow of the hypermedia framework.

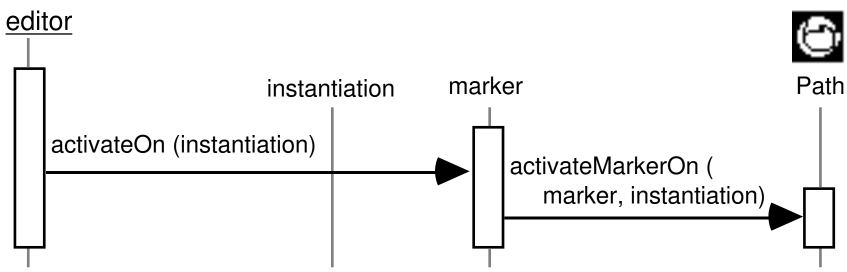
Solution

Define an infinite loop, with a body that defines how control is transferred from one object to another. Incorporate important messages only and leave the detailed protocols are for the contracts. Base the infinite loop on the navigation operation, which is the heart of any hypermedia system. Call it *navigation template*, to emphasise that is a description of a navigation operation omitting details of the participants involved.

Contract

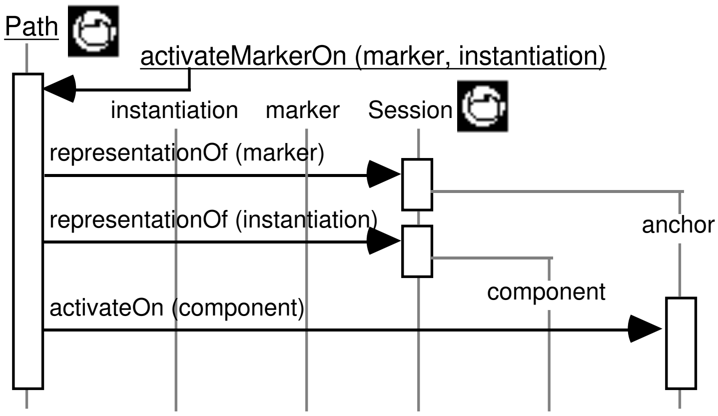
(See [object interaction diagrams] for a short survey of the main elements in an object interaction diagram)

Step 1: Selection of origin point



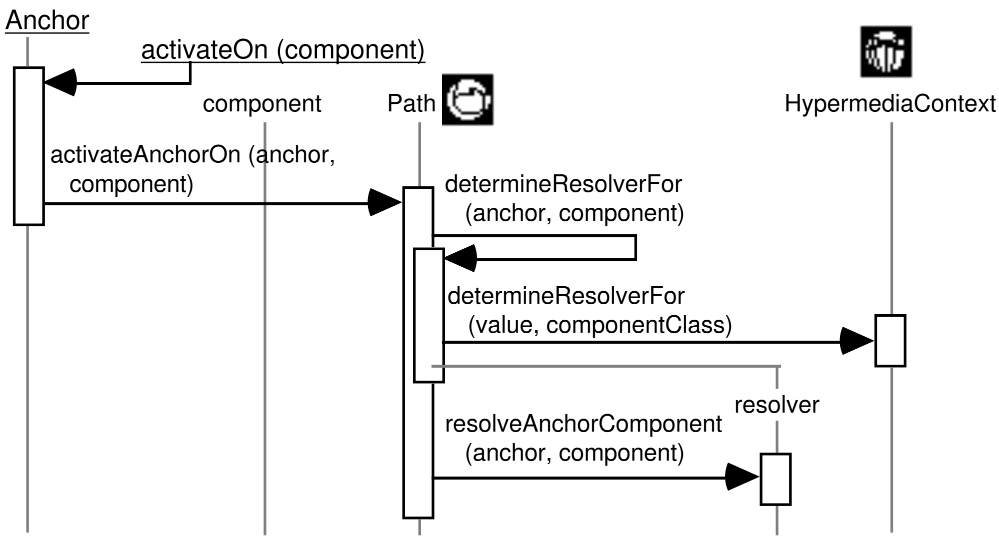
All navigation starts with a selection of the navigation source by the viewer application. This event is detected in the editor object and initiated by sending the `activateOn` message to a marker with an associated instantiation object as parameter. The marker transfers control to the Path meta-object by means of the `activateMarkerOn` message.

Step 2: Identification of origin point



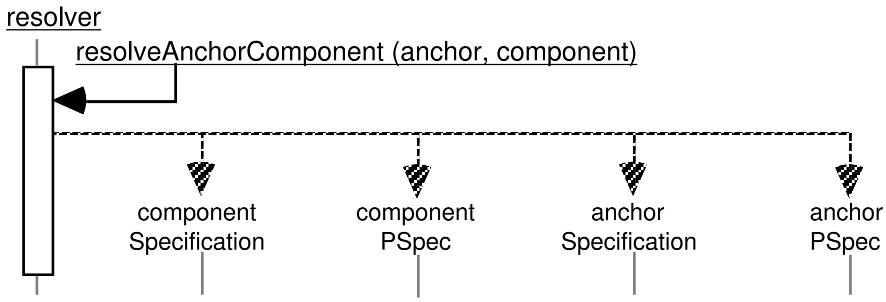
On receiving the `activateMarkerOn` message, the Path infers the anchor and component associated with the given marker and instantiation by sending the `representationOf` message to the Session meta-object. Afterwards, the Path transfers control to the anchor object by means of the `activateOn` message (with the component as parameter). Note that this scheme allows to start navigation without presentation layer objects, which is important for automatic analysis of static hypermedia structures.

Step 3: Start resolution process



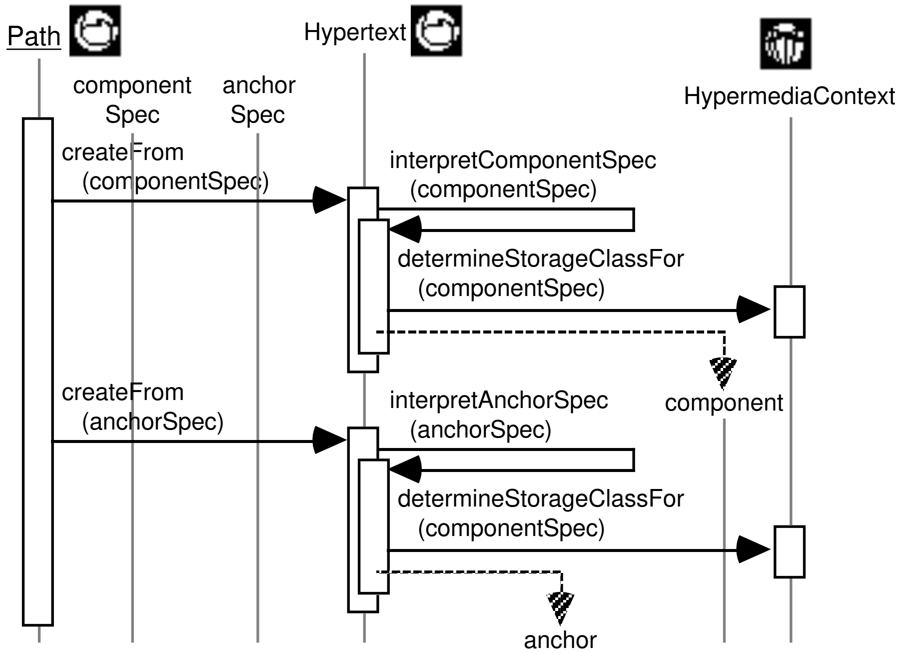
On receiving the `activateOn` message, the anchor transfers control back to the Path meta-object by means of the `activateAnchorOn` message. The Path then identifies the resolver that is supposed to handle the resolution request by sending the self message `determineResolverFor`. As part of this identification process, the Path requests the HypermediaContext meta-meta-object for the name of an installed and active resolver. Afterwards, the Path requests the identified resolver for the navigation targets by means of the `resolveAnchorComponent` message.

Step 4: Resolution process

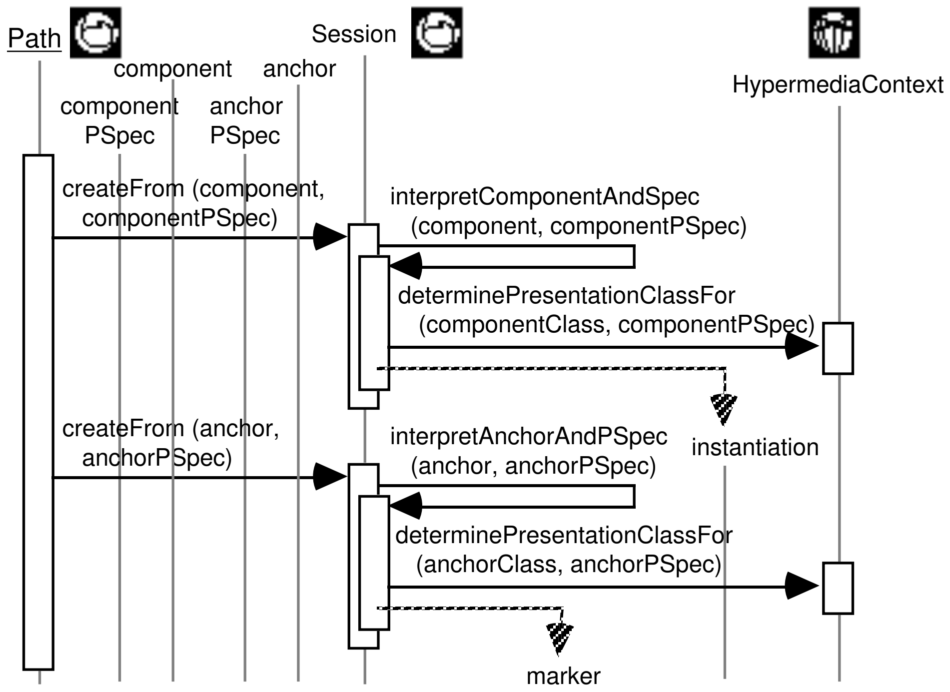


On receiving the `resolveAnchorComponent` request, the resolver returns a collection of quadruples to the Path object. Each quadruple in the collection is a specification for one navigation target.

Step 5: Target identification

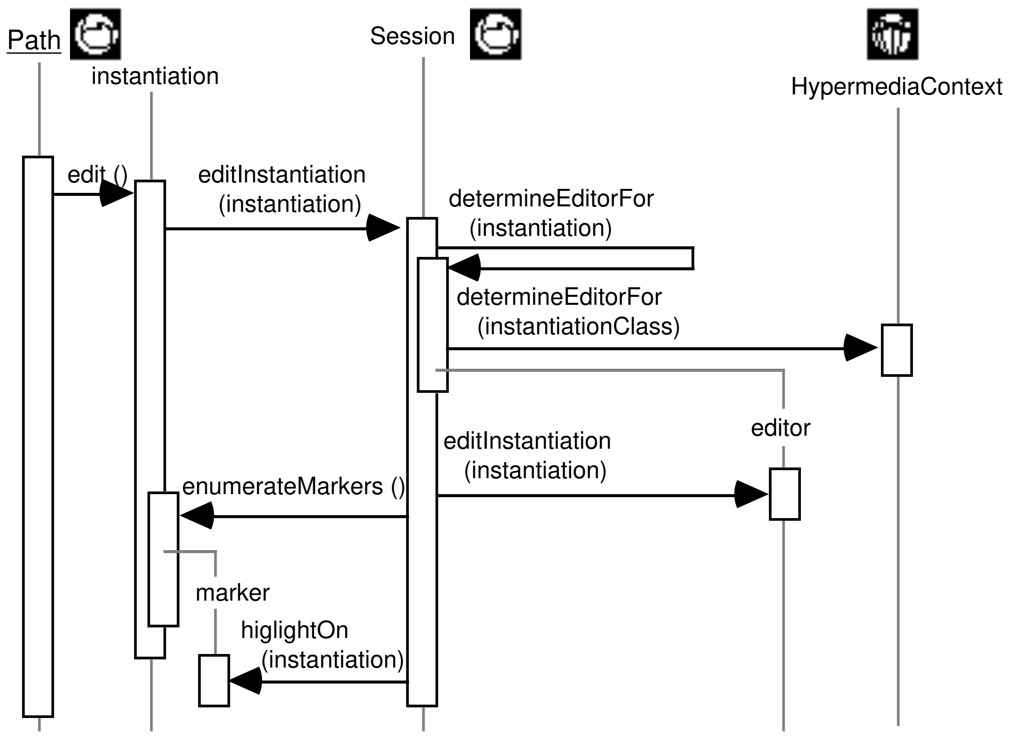


On receiving the quadruples specifying the target locations, the Path turns them into objects representing those locations. For each quadruple, the Path object asks the Hypertext meta-object to create a component (anchor) by sending the `createFrom` message. The Hypertext object interprets that specifier with a self send of `interpretComponentSpec` (`interpretAnchorSpec`), which in turn requests the HypermediaContext object for the class to instantiate by means of the `determineStorageClassFor`.

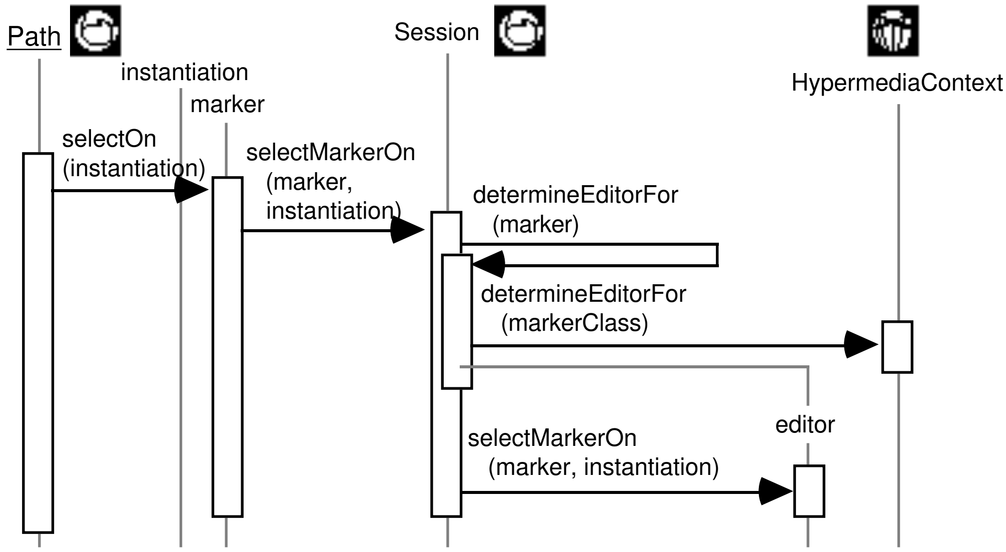


Having created objects representing target locations, the Path uses the rest of the quadruples to turn them into presentable objects. For each component (anchor), the Path object asks the Session meta-object to create an instantiation (marker) by sending the `createFrom` message. The Session object interprets that specifier with a self send of `interpretComponentAndPSpec` (`interpretAnchorAndPSpec`), which in turn requests the HypermediaContext object for the class to instantiate by means of the `determinePresentationClassFor`.

Step 6: Target presentation



Having created the target instantiation object, the Path requests to present it by sending the `edit` message. On receiving the `edit` message, the instantiation passes it on to the Session meta-object. The Session first identifies the editor that is supposed to handle the request by sending the self message `determineEditorFor`. As part of this identification process, the Session requests the HypermediaContext meta-meta-object for the name of an installed and active editor. Afterwards, the Session transfers control to the identified editor by means of the `editInstantiation` message. On returning the Session enumerates all markers associated with an instantiation (by sending the `enumerateMarkers` message to the instantiation object) and highlights them by sending the `highlightOn` message. Note that the `highlightOn` message is processed like the `selectOn` message.



Having created the target marker object, the Path requests to select it by sending the `selectOn` message. On receiving the `selectOn` message, the marker passes it on to the Session meta-object. The Session first identifies the editor that is supposed to handle the request by sending the self message `determineEditorFor`. As part of this identification process, the Session requests the HypermediaContext meta-meta-object for the name of an installed and active editor. Afterwards, the Session transfers control to the identified editor by means of the `selectMarkerOn` message.

Step 7: Back to step 1

After the editor has displayed the instantiation and highlighted the associated markers it goes in sleeping mode, waiting for an event that activates the following navigation action.

Issues

Singleton Meta-Objects

The navigation template is a specification for the control flow in the Zypher open hypermedia framework. We see that a large part of the internal communication in the framework is message passing between the base level objects and meta-level objects. An obvious question is of course "What about performance" ?

It is clear that extra meta-level communication is an overhead compared to systems without a meta-level, thus that a meta-level has a bad effect on the performance. One may argue that this is reasonable: a meta-level is an extra abstraction layer which makes a system more adaptable but that adaptability comes at a cost. Although this is a valuable argument, it is not satisfying because it does not show the whole picture.

Reviewing the diagrams in the navigation template, we see that all the meta-level objects are presented as global objects and that all important messages are funnelled through these global objects. Software engineers recognise this as a "bottle-neck" pattern and know that bottle-necks usually have a bad effect on system's performance.

Yet, there is no rule that says that a meta-level object should be one global object. For instance, if one wants to set-up a system for a distributed environment then it is no use to have a central meta-object that must be consulted for each important activity in the system. To reduce network communication, a better approach would be to replicate the meta-level objects on each client system. In fact, replication is one example that illustrates the value of having a separate meta-level.

Consider a tele-presence situation, where two people are simultaneously editing a shared document and want to be aware of the modifications made by the other. Tele-presence can be achieved by adapting the editor software to pass all open, close and change events to the other application. Such an adaptation implies that one must change the internal implementation of the editor software to incorporate the message passing mechanism. If we review the Zypher design, we see that there is an event-passing mechanism available (see [events]) and that this event-passing is controlled by the meta-level objects (see [meta-objects], [meta-meta-objects]). This architecture allows to replicate the meta-level objects over the two machines and it is feasible to plug in special meta-objects that pass open- and close-events to all other meta-objects residing on other machines. This way, meta-level objects can detect other meta-level objects controlling the same document and synchronise with them by passing change-events. The base level objects remain unchanged, since the event-passing mechanism is still the same.

We conclude that a meta-level would allow to implement the tele-presence example without modifying the internal implementation of the editor software. Although we did not perform the actual experiment, other work has shown that this is feasible approach. CodA [McAffer'95] has been used to open up the implementation of Smalltalk message passing and is able to add meta-level infrastructure to Smalltalk objects so that additional behaviour like concurrency or distribution can be added when necessary. The CodA experiment is especially important as it shows that a meta-level is "just another application" and that traditional software design techniques like abstraction and decomposition remain valuable. Abstract Communication Types [AksitEtAl'93] are objects that abstract interactions among objects and can be used to model distribution and concurrency. Unlike CodA, the abstract communication types are intended to be part of the design of a new programming language (i.e. Sina). Apertos [Yokote,Teraoka,Tokoro'89], [Yokote'92] is object-oriented operating system for mobile computing, supporting distribution and object migration by means of its meta-object protocol. SOM, the System Object Model part of IBM's OS/2 operating system [Forman,Danforth,Madduri'95] includes the notion of meta-classes, which can be used to wrap additional behaviour (locking, notification) around base level message sends.

Relations

Where To Go Next ?

∇ People that read the Zypher design pattern documentation sequentially came at the end. The obvious next step is to review the design by following a non-sequential path.

Other Catalogues

The Path, Session and Hypertext meta-objects and the HypermediaContext meta-meta-object are singletons [GammaEtAl'93]. This allows to implement them as global variables, accessible from the whole system, yet with a restricted access.

PART 3

Conclusion & Appendices

CHAPTER **9**

Conclusion

Wrap Up

The dissertation has started with an analysis of the domain of open hypermedia systems, based on an overview of major milestones in hypermedia research, an examination of a requirement list for open hypermedia systems, a scenario for using open hypermedia systems in an engineering environment and the flag taxonomy for open hypermedia systems. Next we have presented our personal perspective on open hypermedia systems based on the notion of a design space and the puppet master metaphor. The former represents an open hypermedia system as a tailorable volume in a three dimensional axes system — storage, presentation and navigation. The latter identifies three levels of tailorability — domain level, system level and configuration level. We have shown that the notion of a design space with three levels of tailorability covers the key requirements for the design of open hypermedia systems, and —more importantly— that it can be generalised for any other open system.

Next, we have investigated object-oriented frameworks and meta-object protocols, two state of the art techniques proposed by the object-oriented software engineering community.
The concept behind an object-oriented framework is that clients reuse the framework design by supplying concrete components for the open ends of the framework. The open ends of the framework are specified using template methods. A framework comes with a number of contracts: sets of rules telling how framework components should be assembled to work properly. Basically, there exist two approaches for specifying reusable designs: the white-box approach based on class inheritance and the black-box approach based on object composition. In general —for the important design decisions in a framework— the black-box approach is favoured over the white-box approach, because object composition allows to reconfigure the framework at run-time.
A meta-object protocol is an abstraction technique where the interface of a module in an object-oriented framework is split in two separate parts: the base level interface and the meta-level interface. The base level interface is used to access the basic functionality of the module, while the meta-level interface allows us to control designated implementation aspects of that module. Experiments have shown that meta-object protocols make systems more adaptable. However, it is recognised that the design of a meta-object protocol remains an open question. First, because it is hard to recognise when something is part of the meta-level interface; second —and this is a fundamental problem—, because it is too expensive to develop meta-level interfaces incrementally. We have coped with the first problem by adopting a meta-level criterion based on the notions of 'about-ness' and 'causal connection'. For the second problem, we have proposed a methodology based on a proper combination of object-oriented frameworks and meta-object protocols.

Our methodology is based on the insight that explicit representations of framework contracts are part of the meta-level, which leads us to the conclusion that *the design of a framework provides an initial ground for the design of a meta-object protocol.* As a consequence we can be confident that the design of the meta-level interface can reuse parts of the design of the framework's base level interface. This is especially important as lots of promising techniques for designing base level interfaces are under way.

The methodology we propose is summarised in four design guidelines

1) Devise a design space for the problem domain.
2) Each design space axis should correspond to a black-box template method.
3) Each design space axis should correspond to a framework contract.
4) The configuration of the design space axes should correspond to a framework contract.

We claim that by following these guidelines, it is possible to build a system for the intended design space that incorporates the three levels of tailorability: domain level, system level and configuration level. As these three levels of tailorability cover key requirements for open systems, our methodology is well suited for the design of open systems in general.

The second half of the dissertation presents an experimental validation of our methodology in the domain of open hypermedia. It provides a description of the so-called Zypher open hypermedia framework as a design pattern language, which is a state of the art technique for describing framework design.

The design space mentioned in the first design guideline is the three dimensional hypermedia design space —storage, presentation, navigation— that resulted from our analysis of the domain of open hypermedia systems.

The initial design of our framework builds on the design guidelines summarised in the flag taxonomy for open hypermedia systems. As such, our design incorporates a separation between storage and presentation on the one hand and a separation between contents and structure on the other hand. Although the flag taxonomy provided us with a good factorisation of the problem domain, it enforces a white-box approach for incorporating the hypermedia design space. To allow run-time reconfiguration, we applied our second design guideline by extending the design with black-box template methods for each of the axes in the design space. This way, we show that *domain level tailorability corresponds with the base level of a framework.*

In a next step, we incorporate system level tailorability in the design of our framework. Following our third design guideline, we see that this corresponds with wrapping additional behaviour around the execution of the framework contract. Favouring a black-box approach, we extend the design to incorporate an object that is an explicit representation of such a framework contract. As we know that an explicit representation of a framework contract is a meta-object, we illustrate that *system level tailorability corresponds with a meta-level of a framework.*

In a final step, we incorporate configuration level tailorability; a process that is very similar to the previous step. Based on our fourth design guideline, we see that this corresponds with changing the execution of the configuration contract and again we extend the design with an explicit representation of that contract. The fundamental difference with the system level tailorability extension, is that the latter extension is part of the meta-level of the meta-level. So we conclude that *configuration level tailorability corresponds with a meta-meta-level of a framework.*

Besides providing the experimental validation for the proposed design methodology, the Zypher framework serves a second purpose: its abstract design was made concrete in a working prototype of a so-called framework browser. A framework browser is a software system that supports software engineers in the management of complex object-oriented architectures, by providing a seamless integration between the design and the implementation of frameworks. This framework browser has actually been used to manage the relationships between the Zypher design pattern documentation and the source code of the implementation. For example, it is possible to navigate between design pattern documentation displayed using a third-party text editor and the framework implementation displayed using the programming tools that are part of the Smalltalk programming environment. Also, we are able to generate the necessary structures to make this integrated environment accessible over the world-wide web.

The Research Hypotheses Revisited

In the introduction we have formulated three hypotheses. With all of the above in mind, it is time to reconsider the hypotheses and comment on their appropriateness.

Research Hypothesis 1

> *Object-oriented frameworks and meta-level abstraction are two complementary techniques in the design of open systems.*

As noted, no-one doubts whether object-oriented frameworks and meta-level abstractions are useful in the design of open systems as both techniques have already found their way into the design of commercial systems. So the hypothesis boils down to the question how the two techniques can complement one another.

Our work has shown that explicit representations of framework contracts are meta-objects, which implies that a potential combination of both techniques consists of a framework design with explicit objects representing framework contracts. At least this shows that the design of the base level provides clues for the initial design of the meta-level, which is an important insight in its own right.

However, this says nothing about the contracts that should be made explicit in the design. Obviously, we need some support to tell what contracts should be made explicit. Here we have proposed the four design guidelines based on the notion of a design space. Once such a design space has been devised, the four guidelines help a framework designer to identify the contracts that should be included in the meta-level. This way it is possible to incorporate the notions of domain level, system level and configuration level tailorability into the design of the framework. Moreover, we have shown that domain level tailorability can be achieved with an object-oriented framework, while system and configuration level tailorability can be achieved by meta-level abstraction. So, we conclude that both techniques are complementary in the design of open systems: *they share the common goal of tailorability, yet they differ in the level of tailorability that can be achieved.* Assuming that the three levels of tailorability correspond to an open system (see the explanation with the second research hypothesis), we claim that our first research hypothesis remains appropriate.

Research Hypothesis 2

> *Open hypermedia systems are representative for the larger set of open systems.*

The second research hypothesis was formulated because it allows us to generalise the results obtained within the context of an open hypermedia software artefact for the generic set of open systems — examples of other kinds of open systems being operating systems, database systems, inter application communication standards, tailorable software systems and programming languages.

To support the appropriateness of the hypothesis, we have argued that open hypermedia covers themes that recur in all these kinds of open systems —interoperability, extensibility and distribution. Also, we have shown that the three levels of tailorability cover those recurring themes, so we conclude that the three levels of tailorability are appropriate in the context of other kinds of open systems as well.

Then there must be a way to transpose the results of the open hypermedia software artefact to other kinds of open systems. Here we have emphasised that the definition of the three levels of tailorability is based on the notion of a design space, but is completely independent of the type and number of axes in that design space. So we conclude that, *if it is possible to devise a design space for a particular open system, then it is possible to transpose the techniques to incorporate the three levels of tailorability*. Based on other experiments (i.e., Agora and

ApplFLab) we are confident that devising a design space is feasible in other domains as well. So, assuming that it is possible to devise a design space for the application domain of the target open system, we claim that our second research hypothesis remains appropriate.

Research Hypothesis 3

Open hypermedia is a technological cornerstone for software engineering.

The third research hypothesis causes some kind of a feedback loop, in the sense that the first and second research hypotheses are about the potential benefits of applying software engineering techniques to the domain of open hypermedia systems, while the third hypothesis deals with the advantages of applying open hypermedia technology to manage the complexity of software engineering.

We have conducted various experiments in the context of what we have been calling a framework browser and we have achieved some encouraging results. First, we have shown that it is possible to use open hypermedia to connect third-party applications with the programming environment, which enables us to link source-code with design documentation in all kinds of formats. Second, we have experimented with a componentware approach for composing specific code browsers, which is advantageous for the construction of design specific views on the implementation. Third, we have examined the possibilities of an extensible link engine by incorporating specific algorithms to infer navigation relationships, which is helpful in consistency maintenance of large dynamic structures. Finally, we have explored open hypermedia as a way to publish reusable designs on the world-wide web, which is valuable in the context of reuse libraries.

The outcome of our experiments is encouraging, but there is a lot of work to be done to build a true framework browser: one that provides a seamless integration between the numerous aspects of framework development. Also, lots of ideas remain unexplored, especially concerning the support for teams. As far as we stand now, we are not able to assert the appropriateness of the third research hypothesis. Nevertheless, none of the experiments contradicted it, so we conclude that *the application of open hypermedia technology in the context of software engineering remains an interesting area for future work.*

Open Questions

Having clarified and refined some important issues in both object-oriented software engineering and open hypermedia, it is time to consider some unanswered questions and point at known limitations of our approach.

System Level Tailorability

System level tailorability is about wrapping additional behaviour around points on a design space axis and the kind of behaviour we aim at is concurrency control, logging, caching, authority control and integrity control. We propose a meta-level interface which allows us to wrap additional behaviour around the execution of framework contracts without affecting the internal implementation of the contract participants.

The typical example is maintaining a trail of all locations visited in a hypermedia session. This corresponds with wrapping logging behaviour around the execution of the navigation contract; i.e. the contract between the anchor and the resolver (see figure 23). By introducing the path object as an explicit representation of that navigation contract we are able to plug in logging behaviour without affecting the internal implementation of the anchor or resolver objects.

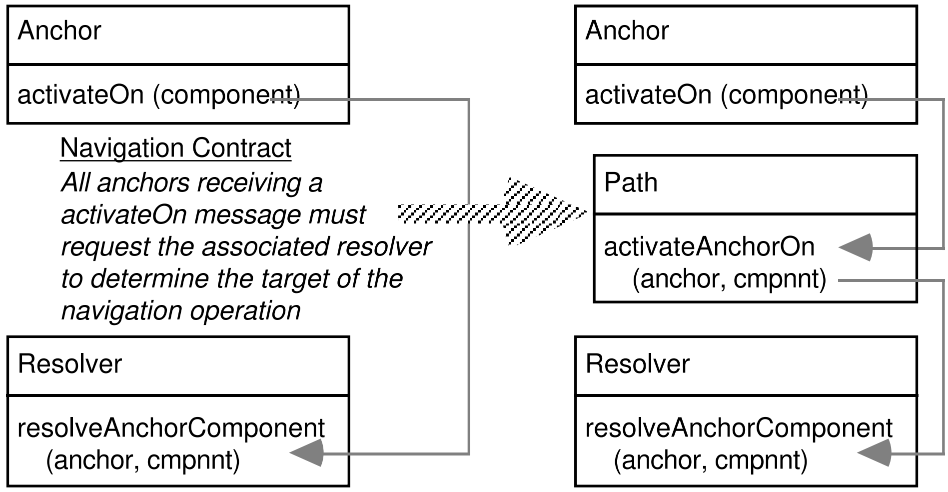


Figure 23: Logging the Navigation Contract

Actually, it is not entirely correct that the participating objects remain unaffected. Indeed, besides trapping all navigation operations one must also be able to memorise the internal state of the anchor and the component, to restore it for later use. Since both the anchor and component may represent virtual data, memorising the internal state is a complex matter. Nevertheless, the problem can be solved by applying the memento design pattern [GammaEtAl'95], which requires an extension of the interface of both the anchor and resolver objects. Such an extension may imply that one must affect the internal implementation of the participating objects, i.e. by adding extra instance variables.

Similar patterns are necessary to implement services like concurrency control, caching, authority control and integrity control; i.e. to wrap additional behaviour around an occurrence of a contract, one may need to extend the interface of the participants of that contract. However, those extensions don't have anything to do with the actual contract and are in fact situated at some other level. That is, such interface extensions must deal with issues like object identifiers, internal state, time stamps, …, which are issues that have more to do with the machine the system is running on, or the programming language the system is implemented with. Normally, such functionality is inherited from the root classes of the inheritance hierarchy.

So we must relax our assertion in the sense that one can incorporate system level tailorability by having a meta-level interface that allows to change the system behaviour, but sometimes one must make slight modifications to the external interface of base level objects. Nevertheless, the modifications to the external interface are independent compared to the extra services created and should be available for a complete session, so specified at system configuration time. In such a situation, the white-box approach is appropriate, so one can implement the extra functionality in the root classes of the inheritance hierarchy and reuse them for many kinds of services.

Singleton Meta-Objects

Another intriguing question is whether the meta-objects should remain singletons —as they are in the design of the Zypher framework— or if it is a good idea to allow multiple instances of them. There are cases supporting the viewpoint that it is better to have multiple instances.

First of all, consider a distributed setting. There, it is definitely not a good idea to work with a singleton meta-object, as such an object corresponds to a bottle-neck for the important events in the global control flow. To eliminate the bottle-neck, one may replicate the meta-object, so that there is one instance of the meta-object for each component in the distributed architecture.
So a distributed setting requires multiple instances of a meta-object. Note that all those instances will differ in internal state, but that their behaviour is essentially the same.

One may ask whether there exist situations that require multiple instances of meta-objects, where those instances will differ in both state and behaviour. Considering the schema in figure 23, suppose one wants to log all activations of anchors created after a certain date and leave the activations of all other anchors out of the log. One solution would be to include some filter in the log algorithm implemented in the path object. Another solution would be to have two path objects with different behaviour; one that writes its activations to the log and another one that does not. The filter is then embedded in a separate dispatcher object that decides what objects should be managed by the first path object, and what should be managed by the second one.
Although the first solution is easier to implement, the second solution is more flexible as it corresponds to a black-box approach. For example, as the filter algorithm is independent of the path object, the second solution makes it possible to reconfigure the logging algorithm at run-time.

Having multiple instances of meta-objects that differ in both state and behaviour is more flexible, yet this flexibility comes at a cost. As long as there is only one meta-object —or multiple replicas of one and the same object—, one can maintain a reference to this meta-object in some global variable. Once there exist multiple instances that differ in both state and behaviour, one must include some reference mechanism from the base level object to the meta-object that is supposed to handle the meta-level behaviour. Such reference mechanism may be quite complicated, as it may depend on a particular combination of base level objects. For example, the path meta-object may differ depending on the particular anchor-component-resolver triple that participates in the navigation contract.

Additional Levels of Tailorability

We have proposed three levels of tailorability —domain level, system level and configuration level— and have argued that they correspond to respectively a framework base level, a meta-level and a meta-meta-level. The levels of tailorability covered the open hypermedia requirement list, which implies that we need at least those three levels of tailorability. However, one can ask if there are other useful levels of tailorability and whether they would correspond with higher meta-levels as well.

We don't have an answer to this question, but we do recognise two viable candidates for additional levels. A first one is about adding or removing axes to a design space. As we claim that the levels of tailorability are independent of the type and number of axes, it should be possible to include an interface to change the axes system. And somehow this would correspond with a higher meta-level, because it must necessarily change the meta-meta-object representing the configuration of the design space.

A second one is about the configuration of the meta-objects. In the current design, there is only one instance of each meta-object in the system, but —as argued above— some situations may require multiple instances of the same meta-object. And then the decision about what instance of a meta-object should handle what base level object is crucially important. The influence this decision an additional level of tailorability is required. This would correspond with a higher meta-level, because it is about the configuration of the meta-objects.

Additional Hypermedia Design Space Axes

We proposed our perspective on hypermedia as a three dimensional design space, the three dimensions being storage, presentation and navigation. We argued that the name and number of the dimensions does not affect the remainder of our argumentation, this way avoiding the discussion if those three dimensions are adequate for describing the space of hypermedia systems. In fact, it remains an open question whether the three dimensions are necessary and sufficient.

We are quite confident that a hypermedia design space should at least include those three dimensions. First, because all the dimensions are somehow included in the hypertext & hypermedia definition (p.08) we adopted; second, because the experiments with the framework browser have shown that the three dimensions are at least appropriate. Nevertheless, we don't have any arguments for supporting whether the three dimensions are sufficient.

An interesting candidate for an extra dimension would be the authoring of navigation structures. Authoring is an aspect that is not made explicit in our design, because we have focused on navigation structures that can be derived from the contents. So, in the Zypher prototype, authoring is part of the navigation dimension, which implies that each point on the navigation axis (i.e. each resolver) defines its own authoring facilities. An experiment that may provide some answers would be to capture a hypermedia system that emphasises on the authoring of hypermedia structures —such as the spatial hypermedia systems Aquanet and VIKI— in the current design of the Zypher framework.

Black-box versus White-box

The argumentation for incorporating the three levels of tailorability into the design of a framework was about favouring a black-box approach over a white-box approach, or in other words object composition over class inheritance. One may ask what would happen if for some reason the white-box approach was favoured over the black-box approach.

Consider the navigation axis of our hypermedia design space (see also figure 23). To incorporate domain level tailorability, we need to be able to vary the resolution of the anchor independently of its activation, which implies that the second design guideline would be rephrased as "each design space axis should correspond to a white-box template method". In the white-box reuse approach, this would mean that we have a single anchor object that understands both the `activateOn` and `resolveAnchorComponent` messages, but that both of them may be inherited from different classes.

To incorporate system level tailorability on the navigation axis, we must be able to wrap additional behaviour around the execution of the `activateOn` and `resolveAnchorComponent` methods. In a white-box approach this means that we must inherit the wrapping behaviour from a common superclass of the anchor classes or the resolver classes involved. To ensure this inheritance behaviour in the framework, the framework must include a contract stating that anchor classes and resolver classes must delegate the navigation messages to their superclass; i.e. they must perform a super send. So the third design guideline, stating that "each design space axis should correspond to a framework contract" remains valid, but the way this design guideline is interpreted in the design of the framework is different.

To incorporate configuration level tailorability in a white-box approach, we must be able to extend the inheritance hierarchy of the framework; preferably using multiple inheritance. For instance, if we want to have a relationship between a particular point on the navigation axis and a particular point on the storage axis, we will create an anchor class that inherits from a class implementing the behaviour of that storage device and a class that implements that particular link resolution algorithm. However, in that case the inheritance hierarchy becomes an essential aspect of the framework design, which implies that it should be captured in a framework contract. So, the fourth design guideline, stating that "the configuration of the design space axes should correspond to a framework contract" remains valid, but again the way this design guideline is interpreted in the design of the framework is different.

The main argument for favouring the black-box approach over white-box approach is that this makes it easy to reconfigure the framework at run-time: one can change the execution of black-box template methods by changing the object composition. In a white-box approach, run-time reconfiguration of the framework corresponds with run-time modifications to the inheritance hierarchy. There exist object-oriented languages (Smalltalk and CLOS to mention the most popular ones) that allow to modify the class hierarchy at run-time, however this functionality always belongs to the meta-object protocol of the programming language. Indeed, the class hierarchy is about the software system in a causally connected way.

Note that the white-box approach —even with the possibility of run-time framework reconfiguration— remains less flexible than the black-box approach, because the black-box approach allows to reconfigure the framework at object level, while the white-box approach is restricted to reconfigurations at the class level. Reconfiguration at object level in a white-box approach would imply the possibility of changing the class of an existing object at run-time which is a very difficult operation at it requires changes to the objects internal state representation (i.e. the set of instance variables) and behaviour (i.e. the method table). Anyway, such an operation would also belong to the meta-object protocol of the programming language, as the object-class relationship is about the software system in a causally connected way.

The above argumentation leads us to the conclusion that favouring a white-box approach over a black-box approach would not change that much. To incorporate the three levels of tailorability, the design guidelines we propose remain essentially the same; only the way they should be interpreted in the design of the framework changes. And if one wants to reconfigure the framework at run-time, we need a meta-object protocol that is part of the programming language used to implement the framework.

Afterthought

As an final remark, we emphasise on the cross-fertilisation embodied in our work. On the one hand we build on the accumulated experience of the software engineering community. More precisely we deal with techniques studied in very technological areas like programming languages, distributed architectures and operating systems. As a consequence, the techniques are well specified and their effects quite accurate, yet they are seldom applied in other applications domains. On the other hand, our work builds on the insights of hypermedia, a domain with years of fruitful research on how to enhance the way people deal with information. The problems encountered there are rather fuzzy and the solutions quite imprecise, yet hypermedia research has changed the working habits of almost all people working with computers.

Our contribution is original in the way it captures the best of both worlds: the precision of software engineering with the applicability of hypermedia. And this idea reveals the heart of what we entitled "the link from object-oriented software engineering to open hypermedia".

CHAPTER *10*

Appendices

References

[Abelson,Sussman'84]
Abelson, H. / Sussman, G. J. "Structure and Interpretation of Computer Programs"; MIT Press, 1984.

[Akscyn'91]
Akscyn, R. "Design Tradeoffs for Advanced Hypertext Technology"; Tutorial Notes at [HT'91].

[Akscyn,McCracken,Yoder'87]
Akscyn, R. / McCracken, D. / Yoder, E. "KMS: A Distributed Hypermedia System for Managing Knowledge in Organisations"; in [HT'87]. Republished in Communications of the ACM - Vol. 31(7) - July '88.

[AksitEtAl'93]
Aksit, M. / Wakita, K. / Bosh, J. / Bergmans, L. / Yonezawa, A. "Abstracting Object Interactions Using Composition Filters"; ECOOP '93, Workshop on Object-Based Distributed Programming - Lecture Notes in Computer Science, nr. 791 – Berlin Heidelberg, July 1993. Available on the world-wide web via"ftp://ftp.cs.utwente.nl/pub/doc/TRESE/AbstrObjIntUsingCF.ps.Z".

[Anderson,Taylor,Whitehead'94]
Anderson, K. M. / Taylor, R. N. / Whitehead, E. J. Jr. "Chimera: Hypertext for Heterogeneous Software Environments"; in [ECHT'94].

[Andrews,Kappe,Maurer'95]
Andrews, K. / Kappe, F. / Maurer, H. "Serving Information to the Web with Hyper-G"; Proceedings of the Third International world-wide web Conference - WWW'95; Computer Networks and ISDN Systems - Vol. 27(6) - April 1995.

[Apple'89]
Apple Computer, Inc. "Macintosh HyperCard User's Guide"; Apple Computer, Inc., 1989.

[Ashman'94]
Ashman, H. "Opinion: What is Hypermedia"; ACM SIGLINK Newsletter - Vol. 3(2) - September '94.

[Aßfalg'94]
Aßfalg, R (Ed) "Open Hypertext Systems" at the Universität Konstanz, 26/27 May 1994. Send e-mail to Rainer Kuhlen (kuhlen@inf-wiss.uni-konstanz.de) or Marc Rittberger (ritt@inf-wiss.uni-konstanz.de).

[Beck,Johnson'94]
Beck, K. / Johnson, R. "Patterns Generate Architectures"; ECOOP'94 Proceedings, Lecture Notes in Computer Science nr. 821, Springer-Verlag, 1994. Available on the world-wide web via http://st-www.cs.uiuc.edu /users /patterns /patterns.html.

[Berners-LeeEtAl'94]
Berners-Lee, T. / Cailliau, R. / Luotonen, A. /

Nielsen, H. F. / Secret, A. "The world-wide web"; Communications of the ACM - Vol. 37(8) - August '94.

[Birrer,Eggenschwiler'93]
Birrer, A. / Eggenschwiler, T. "Frameworks in the Financial Engineering Domain: An Experience Report"; ECOOP'93 Proceedings, Lecture Notes in Computer Science nr. 707, Springer-Verlag, 1993.

[Blaha,Premerlandi,Rumbaugh'88]
Blaha, M. R. / Premerlani, W. J. / Rumbaugh, J. E. "Relational Database Design Using an Object-Oriented Methodology"; Communications of the ACM - Vol. 31(4) - April '88.

[Brooks'87]
Brooks, F. P. Jr. "No Silver Bullet: Essence and Accidents in Software Engineering"; IEEE Computer - April '87.

[Bush'45]
Bush, V. "As we May Think"; Atlantic Monthly - 176 - July, 1945. Available on the world-wide web via http://www.isg.sfu.ca/~duchier/misc/vbush/.

[Campbell,Goodman'87]
Campbell, B. / Goodman, J. M. "HAM: A General - Purpose Hypertext Abstract Machine"; in [HT'87]. Republished in Communications of the ACM - Vol. 31(7) - July '88.

[CampbellEtAl'93]
Campbell, R. H. / Islam, N. / Raila, D. / Madeany, P. "Designing and Implementing Choices: An Object-oriented System in C++"; Communications of the ACM - Vol. 36(9) - September'93.

[Catlin,Bush,Yankelovich'89]
Catlin, T. / Bush, P. / Yankelovich, N. "InterNote: Extending a Hypermedia Framework to Support Annotative Collaboration"; in [HT'89].

[CodenieEtAl'94]
Codenie, W. / De Hondt, K. / D'Hondt, T. / Steyaert, P. "Agora: Message Passing as a Foundation for Exploring OO Language Concepts"; ACM SIGPLAN Notices, Vol. 29(12), December '94.

[Conklin'87]
Conklin, J. "Hypertext: An Introduction and Survey"; IEEE Computer - Vol. 20 (9) - September 1987.

[Conklin,Begeman'88]
Conklin, J. / Begeman, M. L. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion"; CSCW'88 Proceedings, ACM Press, 1988.

[Cotter,Potel'95]
Cotter, S. / Potel, M. "Inside Talingent Technology"; Addison-Wesley, 1995.

[Davis'95]
Davis, H. "To Embed or Not to Embed …"; Communications of the ACM - Vol. 38(6) - August '95. Available on the world-wide web via http://www.acm.org/siglink/.

[Davis,Knight,Hall'94]
Davis, H. C. / Knight, S. / Hall, W. "Light Hypermedia Link Services: A Study of Third Party Application Integration"; in [ECHT'94].

[Davis,Lewis,Rizk'96]
Davis, H. / Lewis, A. / Rizk, A. "OHP: A Draft Proposal for a Standard Open Hypermedia Protocol"; in [Wiil,Demeyer'96]

[DavisEtAl'92]
Davis, H. / Hall, W. / Heath, I. / Hill, G. / Wilkins, R. "Towards An Integrated Information Environment with Open Hypermedia Systems"; in [ECHT'92].

[DeBra,Houben,Kornatzky'92]
De Bra, P. / Houben, G. J. / Kornatzky, Y. "An Extensible Data Model for Hyperdocuments"; in [ECHT'92].

[DuvalEtAl95]
Duval E. / Olivie H. / O'Hanlon P. / David G. Jameson "HOME: an Environment for Hypermedia Objects"; Journal of Universal Computer Science - Vol 1(5) - May 1995. Available on the world-wide web via http://www.iicm.tu-graz.ac.at/Cjucs_root. Also "HOME: Hypermedia Object Management Environment"; Demonstration at ECHT'94.

[ECHT'90]
European Conference on Hypertext'90, November 27-30, Paris, France. Proceedings: Rizk, A. / Streitz, N. / André, J. "Hypertext: Concepts, Systems and Applications - Proceedings of the European Conference on Hypertext", Cambridge University Press, 1990.

[ECHT'92]
European Conference on Hypertext'92, November 30 - December 4, Milano, Italy. Proceedings: ACM Press, 1992.

[ECHT'94]
European Conference on Hypertext'94, September 18-23, Edinburgh, UK. Proceedings: ACM Press, 1994.

[Engelbart'95]
Engelbart, D. C. "Toward Augmenting the Human Intellect and Boosting our Collective IQ"; Communications of the ACM - Vol. 38(8) - August '95. Available on the world-wide web via http://www.acm.org/siglink/.

[Forman,Danforth,Madduri'95]
Forman, I. R. / Danforth, S. / Madduri, H. "Composition of Before/After Metaclasses in SOM"; AIXpert, May 1995.

[FountainEtAl'90] Fountain, A. M. / Hall, W. / Heath, I. / Davis, H. C. "MICROCOSM: An

Open Model for Hypermedia With Dynamic Linking"; in [ECHT'90].

[Furuta,Stotts'89]
Furuta, R. / Stotts, P. D. "Programmable Browsing Semantics in Trellis"; in [HT'89].

[GammaEtAl'95]
Gamma, E. / Helm R. / Johnson R. / Vlissides, J. "Design Patterns"; Addisson-Wesley, 1995. The same people have written the paper "Design Patterns: Abstraction and Reuse of Object-Oriented Design"; ECOOP'93 Proceedings, Lecture Notes in Computer Science nr. 707, Springer-Verlag, 1993.

[Goldberg'84]
Goldberg, A. "Smalltalk-80, the Interactive Programming Environment"; Addison-Wesley, 1984.

[Goldberg,Rubin'95]
Goldberg, A. / Rubin, K. S. "Succeeding with Objects. Decision Frameworks for Project Management"; Addison-Wesley, 1995.

[Grønbaek'94]
Grønbaek, K. "Composites in a Dexter-Based Hypermedia Framework"; in [ECHT'94].

[Grønbaek,Malhotra'94]
Grønbaek, K. / Malhotra, J. "Building Tailorable Hypermedia Systems: the embedded-interpreter approach"; OOPSLA '94 Proceedings - ACM Press, 1994.

[Grønbaek,Trigg'94]
Grønbaek, K. / Trigg, R. H. "Design issues for a Dexter-Based Hypermedia System"; Communications of the ACM - Vol. 37(2) - February '94. Also in [ECHT'92].

[Grønbaek,Trigg'96]
Grønbaek, K. / Trigg, R. H. "Towards a Dexter-Based model for Open Hypermedia: Unifying Embedded References and Link Objects"; in [HT'96].

[GrønbaekEtAl'94]
Grønbaek, K. / Hem, J. A. / Madsen, O. L. / Sloth, L. "Systems: A Dexter-based Architecture"; Communications of the ACM - Vol. 37(2) - February '94. Also in [HT'93] under the title "Designing Dexter-based Cooperative Hypermedia Systems".

[HaanEtAl'92]
Haan, B. J. / Kahn, P. / Riley, V. A. / Coombe, J. H. / Meyrowitz, N. K. "IRIS Hypermedia services"; Communications of the ACM - Vol. 35(1) - January '92.

[Halasz'87]
Halasz, F. G. "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems"; in [HT'87]. Republished in Communications of the ACM - Vol. 31(7) - July '88.

[Halasz'91]
Halasz, F. G. "Seven Issues: Revisited"; Keynote Address at [HT'91]. The slides and a transcript of the speech are available on the world-wide web via http://www.parc.xerox.com/spl/projects/halasz-keynote/transcript.html.

[Halasz,Schwartz'90]
Halasz, F. / Schwartz, M. "The Dexter Hypertext Reference Model"; Proceedings of the 1990 NIST Hypertext Standardisation Workshop (January 16-18, Gaithersburg, MD). Republished in Communications of the ACM - Vol. 37(2) - February '94.

[Hall,Hill,Davis'93]
Hall, W. / Hill, G. / Davis, H. "The Microcosm Link Service"; in [HT'93].

[Helm,Holland,Gangopadhyay'90]
Helm, R. / Holland, I. M. / Gangopadhyay, D. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems"; ECOOP/OOPSLA'90 Proceedings, ACM Press, 1990.

[Hill,Hall'94]
Hill, G. / Hall, W. "Extending the Microcosm Model to a Distributed Environment"; in [ECHT'94].

[HT'87]
Hypertext'87, November 13-15, Chapel Hill, North Carolina. Proceedings: ACM Press, 1987. Some of the papers were republished in Communications of the ACM - Vol. 31(7) - July '88.

[HT'89]
Hypertext'89, November 5-8, Pittsburgh, Pennsylvania. Proceedings, ACM Press, 1989.

[HT'91]
Hypertext'91, December 15-18, San Antonio, Texas. Proceedings: ACM Press, 1991.

[HT'93]
Hypertext'93, November 14-18, Seattle, Washington. Proceedings: ACM Press, 1993.

[HT'96]
Hypertext'96, March 16-20, Washington DC. Proceedings: ACM Press, 1996.

[JacobsonEtAl'92] Jacobson, I. / Christerson, M. / Jonsson, P. / Overgaard, G. "Object-Oriented Software Engineering - A Use Case Driven Approach; Addison-Wesley, 1992.

[Java'95]
Java "The HotJava Browser: A White Paper"; available on the world-wide web via http://java.sun.com/.

[Johnson'92]
Johnson, R. "Documenting Frameworks Using Patterns"; OOPSLA'92 Proceedings, ACM Press, 1992. Available on the world-wide web via http://st-www.cs.uiuc.edu /users /patterns /patterns.html.

[Johnson,Foote'88]
Johnson, R. E. / Foote, B. "Designing Reuseable Classes"; Journal of Object-Oriented Programming - Vol 1(2) - February 1988.

[Johnson,Russo'91]
Johnson, R. E. / Russo, V. F. "Reusing Object-Oriented Designs"; University of Illinois Tech Report UIUICDCS 91-1696.

[Kacmar,Leggett'91] Kacmar, C. J. / Legget, J. J. "PROXHY: A Process-Oriented Extensible Hypertext Architecture"; ACM Transactions on Information Systems, Vol. 9 (4), October '91.

[Kiczales'94a]
Kiczales, G. "Why are Black Boxes so Hard to Reuse ? Toward a New Model of Abstraction in the Engineering of Software"; Keynote address at OOPSLA'94. Available on the world-wide web via http://www.parc.xerox.com/spl/projects/oi/towards-talk/transcript.htm.

[Kiczales'94b]
Kiczales, G. (Ed.) "Foil For The Workshop On Open Implementation"; Xerox Parc. Available on the world-wide web via http://www.parc.xerox.com/spl/projects/oi/workshop-94/foil/main.html.

[Kiczales,Rivières,Bobrow'91]
Kiczales, G. / des Rivières, J. / Bobrow, D. G. "The Art of the Metaobject Protocol"; The MIT Press, 1991.

[Knopik,Bapat'94]
Knopik, T. / Bapat, A. "The Role of Node and Link Types in Open Hypermedia Systems" in [Wiil,Østerbye'94].

[Lai,Manber'91]
Lai, P. / Manber, U. "Flying Through Hypertext"; in [HT'91].

[Leggett,Schnase'94]
Leggett, J. J. / Schnase, J. L. "Viewing Dexter with Open Eyes"; Communications of the ACM - Vol. 37(2) - February '94.

[LewisEtAl.'96]
Lewis, P. H. / Davis, H. C. / Griffiths, S. R. / Hall, W. / Wilkins, R. J. "Media-Based Navigation with Generic Links"; in [HT'96].

[Lucas,Steyaert,Mens'95] Lucas, C. / Mens, K. / Steyaert, P. "Typing Dynamic Inheritance, a Trade-Off between Substitutability and Extensibility. Technical Report vub-prog-tr-95-03.

[Maes'87]
Maes, P. "Concepts and Experiments in Computational Reflection"; OOPSLA'87 Proceedings, ACM Press, 1987.

[Malcolm,Poltrock,Shuler'91]
Malcolm, K. C. / Poltrock, S. E. / Schuler, D. "Industrial Strength Hypermedia: Requirements for a Large Engineering Entreprise"; in [HT'91].

[Marshall,Shipman'93]
Marshall, C. C. / Shipman, F. M. "Searching for the Missing Link: Discovering Implicit Structure in Spatial Hypertext"; in [HT'93].

[Marshall,Shipman'95]
Marshall, C. C. / Shipman, F. M. "Spatial Hypertext: Designing for Change"; Communications of the ACM - Vol. 38(8) - August '95. Available on the world-wide web via http://www.acm.org/siglink/.

[Marshall,Shipman,Coombs'94]
Marshall, C. C. / Shipman, F. M. / Coombs, J. H. "VIKI: Spatial Hypertext Supporting Emergent Structure"; in [ECHT'94].

[MarshallEtAl'91]
Marshall, C.C. / Halasz, F. G. / Rogers, R. A. / Janssen, W. C. "Aquanet: A Hypertext Tool to Hold Your Knowledge in Place"; in [HT'91].

[McAffer'95]
McAffer, J. "Meta level Programming with CodA"; ECOOP'95 Proceedings, Lecture Notes in Computer Science nr. 952, Springer-Verlag, 1995.

[Meyrowitz'86]
Meyrowitz, N. "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework"; OOPSLA'86 Proceedings, ACM Press, 1986.

[Meyrowitz'89]
Meyrowitz, N. "The missing link: Why we're all doing hypertext wrong"; in Barrett, E. (ed) "The Society of Text", The MIT Press, Cambridge, 1989.

[Mylonas,Heath'90]
Mylonas, E. / Heath, S. "Hypertext from the Data Point of View: Paths and Links in the Perseus Project"; in [ECHT'90].

[NationalAcademy'94] National Academy of Sciences "Academic Careers for Experimental Computer Scientists and Engineers"; Communications of the ACM, Vol. 37(2), April '94.

[Nelson'87]
Nelson, T. H. "All for One and One for All"; in [HT'87].

[Newcomb,Kipp,Newcomb'91]
Newcomb, S. R. / Kipp, N. A. / Newcomb, V. T. "Hytime: Hypermedia / Time-based Document Structuring Language"; Communications of the ACM, Vol. 34(11), November '91.

[Nielsen'90]
Nielsen, J. "Hypertext & Hypermedia"; Academic Press, 1990.

[NürnbergEtAl'96]
Nürnberg, P. J. / Leggett, J. J. / Schneider, E. R. / Schnase, J. L. "Hypermedia Operating Systems: A New Paradigm for Computing"; in [HT'96].

[Østerbye,Wiil'96]
Østerbye, K. / Wiil, U. K. "The Flag Taxonomy of Open Hypermedia Systems"; Hypertext '96 Proceedings, ACM Press, 1996. See also [Wiil,Demeyer'96].

[Paepcke'90]
Paepcke, A. "PCLOS: stress testing CLOS"; ECOOP/OOPSLA'90 Proceedings, ACM Press 1990.

[Parunak'91]
Parunak, H.V.D. "Don't Link Me In: Set Based Hypermedia for Taxonomic Reasoning"; in [HT'91].

[Pearl'89]
Pearl, A. "Sun's Link Service: A Protocol for Open Linking"; in [HT'89].

[Poltock'96]
Poltrock, S. E. "Keynote address at the 2nd Workshop on Open Hypermedia Systems — Hypertext '96, Washinton D.C." See [Wiil,Demeyer'96].

[Pree'94]
Pree, W. "Design Patterns for Object-Oriented Software Development"; Addison-Wesley 1994.

[Rao'91]
Rao, R. "Implementational Reflection in Silica"; ECOOP'91 Proceedings, Lecture Notes in Computer Science, P. America (Ed.), Springer-Verlag, 1991.

[Remde,Gomez,Landauer'87]
Remde, J. R. / Gomez, L. M. / Landauer, T. K. "SuperBook: An Automatic Tool for Information Exploration — Hypertext ?"; in [HT'87].

[Rizk,Sauter'92]
Rizk, A. / Sauter, L. "Multicard: An Open Hypermedia System"; in [ECHT'92].

[RumbaugEtAl'91]
Rumbaugh, J. Blaha, M. / Premerlani, W. / Eddy, F. / Lorensen, W. "Object-Oriented Modeling and Design"; Prentice Hall, 1991.

[Sawhney,Balcom,Smith'96]
Sawhney, N. / Balcom, D. / Smith, I. "HyperCafe: Narrative and Aesthetic Properties of HyperVideo"; in [HT'96].

[Schackelford,Smith,Smith'93]
Schackelford, D. E. / Smith, J.B. / Smith, F.D, "The Architecture and Implementation of a Distributed Hypermedia Storage System"; in [HT'93].

[Schmucker'86]
Schmucker, K. "Object-Oriented Programming for the Macintosh"; Hayden Book Company, 1986.

[Schütt,Streitz'90]
Schütt, H. / Streitz, N. "HyperBase: A Hypermedia Engine Based on a Relational Data-Base Management System"; in [ECHT'90].

[Smith'96]
Smith, J.B. "Invited statement at the 2nd Workshop on Open Hypermedia Systems — Hypertext '96, Washinton D.C." See [Wiil,Demeyer'96].

[Smith,Smith'91]
Smith, J. B. / Smith, F. D. "ABC: A Hypermedia System for Artifact-Based Collaboration"; in [HT'91].

[Smith,Zdonik'87]
Smith, K.E. / Zdonik, S. B. "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems"; OOPSLA'87 Proceedings, ACM Press, 1987.

[Steyaert'94]
Steyaert, P. "Open Design of Object-Oriented Languages"; Phd. dissertation, Vrije Universiteit Brussel, 1994. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[Steyaert,DeMeuter'95]
Steyaert, P. / De Meuter, W. "A Marriage of Class- and Object-Based Inheritance Without Unwanted Children"; ECOOP'95 Proceedings, Lecture Notes in Computer Science nr. 952, Springer-Verlag, 1995. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[SteyaertEtAl'93]
Steyaert, P. / Codenie, W. / D'Hondt, T. / Lucas, C. / Van Limberghen, M. "Nested Mixin-Methods in Agora"; ECOOP'93 Proceedings, Lecture Notes in Computer Science nr. 707, Springer-Verlag, 1993. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[SteyaertEtAl'94]
Steyaert, P. / De Hondt, K. / Demeyer, S. / De Molder, M. "A Layered Approach To Dedicated Application Builders Based On Application Frameworks"; In Patel D. / Sun, Y. / Patel, S. (Ed) "Proceedings of the 1994 International Conference on Object-Oriented Information Systems"; Springer-Verlag, 1995. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[SteyaertEtAl'96]
Steyaert, P. / De Hondt, K. / Demeyer, S. / Boyen, N. "Reflective Application Builders"; In Chris Zimmermann (Ed) "Advances in Object-Oriented Metalevel Architectures and Reflection"; CRC Press Inc., Boca Raton, Florida, 1996. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[Trigg'96]
Trigg, R. "Hypermedia as Integration: Recollections, reflections and exhortations"; Keynote Address at [HT'96].

[Udell'94]
Udell, J. "Componentware"; BYTE- May '94.

[VanLimberghen'96]
Van Limberghen, M. "Building Frameworks Through Specialisable Nested Objects"; to appear in Tools-USA'96 Proceedings, Prentice Hall, 1996. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[Vanzyl'94]
Vanzyl, A. J. "Open Hypermedia Systems: Comparisons and Suggestions for Implementation Strategies" in [Wiil,Østerbye'94].

[WeinandEtAl'88]
Weinand, A. / Gamma, E. / Marty, R. "ET++, An Object-Oriented Application Framework in C++"; OOPSLA'88 Proceedings, ACM Press, 1988.

[Wiil,Demeyer'96]
Wiil, U. K. / Demeyer, S. (Ed) "Proceedings of the 2nd Workshop on Open Hypermedia Systems - Hypertext'96"; UCI-ICS Technical Report 96-10. Department of Information and Computer Science, University of California, Irvine, CA 92717-3425. See also http://www.iesd.auc.dk/~kock/OHS-HT96/.

[Wiil,Leggett'92]
Wiil, U. K. / Leggett, J. J. "Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems"; in [ECHT'92].

[Wiil,Leggett'93]
Wiil, U. K. / Leggett, J. J. "Concurrency Control in Collaborative Hypertext Systems"; in [HT'93].

[Wiil,Leggett'96]
Wiil, U. K. / Leggett, J. J. "The HyperDisco Approach to Open Hypermedia Systems"; in [HT'96].

[Wiil,Østerbye'94]
Wiil, U. K. / Østerbye, K. (Ed) "Proceedings of the ECHT'94 Workshop on Open Hypermedia Systems"; Technical report R-94-2038 / Institute for Electronic Systems Department of Mathematics and Computer Science - Fredrik Bajers Vej 7 - DK 9220 Aalborg - Denmark. Available on the world-wide web via ftp://ftp.iesd.auc.dk/pub/packages/hypertext/ECHT94-workshop/.

[Wirfs-Brock,Johnson'90]
Wirfs-Brock, R. J. / Johnson, R. E. "Surveying current Research in Object-Oriented Design"; Communications of the ACM - Vol. 33(9) - September'90.

[Wirth'76]
Wirth, N. "Algorithms+Data Structures=Programs"; Prentice Hall,76.

[Wuyts'96]
Wuyts, R. "Class-Management using Logical Queries, Application of a Reflective User Interface Builder"; GRONICS'96 Proceedings of the third Groningen International Information Technology Conference for Students; University of Groningen, 1996. Available on the world-wide web via http://progwww.vub.ac.be/prog/papers/paperquery.

[Yankelovich,Meyrowitz,VanDam'85]
Yankelovich, N. / Meyrowitz, N. / van Dam, A. "Reading and Writing the Electronic Book"; IEEE Computer - October '85.

[Yokote'92]
Yokote, Y. "The Apertos Reflective Operating System: The Concept and Its Implementation"; OOPSLA'92 Proceedings, ACM Press, 1992.

[Yokote,Teraoka,Tokoro'89]
Yokote, Y. / Teraoka, F. / Tokoro, M. "A reflective architecture for an object-oriented distributed operating system"; ECOOP'89 Proceedings, Cambridge University Press, 1989.

[Zellweger'89]
Zellweger, P. T. "Scripted Documents: A Hypermedia Path Mechanism"; in [HT'89].

Index

List of Figures

Detailed Table of Contents

Aposition / Bijstelling

Group decision processes on the world-wide web must be supported by structured communication to improve information exchange and unstructured communication to cope with meta-discourse.

Explanation

With the appearance of the world-wide web [Berners-LeeEtAl'94], people are confronted with a tremendous increase in communication facilities. However, extra communication facilities do not necesarily improve information exchange. This phenomenon can be observed in discussions on bulletin board systems, newsgroups and mail lists: such discussions quickly lose focus and rarely come to a conclusion. If they come to a conclusion, it is because the group managed to *structure its communication process*, for example by arranging a vote.

Group decision support system (GDSS) research is especially concerned with the question how structured communication processes improve decision making. GDSS like GroupSystems [NunamakerEtAl'91], COPE [Eden,Ackermann'92] and DSide [Kenis'95], [KenisEtAl'95] show that by increasing the number of alternatives considered, enhancing the depth of analysis each alternative is considered with and enlarging the participation of all group members the quality of the decision process improves. Nevertheless, studies revealed that individual group members often feel constrained by the structure, among others for not supporting meta-discourse[11] [Conklin,Begeman'88]. Hence the requirement for some kind of *unstructured communication* between participants.

In a synchronous-proximate setting (i.e. all group members gather at the same time, in the same place), unstructured communication is usually accomplished by some form of verbal contact (mostly through the facilitator). However, experiments that abandon the meeting room can not rely on verbal contact and need other unstructured communication channels. During an experiment with the DSide version accessible via the world-wide web [KenisEtAl'96] we found out that participants used e-mail as a natural communication channel for meta-discourse, allthoug e-mail was not an explicit part of the DSide system.

The use of unstructured communication for meta-discource can be observed in other experiments as well. GroupSystems has been tested in a synchronous-remote setting [NunamakerEtAl'91] and their unstructured communication was available under the form of video conferencing. COPE has been tested in an asynchronous-proximate setting [Trahand'93], but members discussed the decision graph afterwards.

[Berners-LeeEtAl'94] Berners-Lee / Cailliau, R. / Luotonen, A. / Nielsen, H. F. / Secret, A. "The world-wide web"; Communications of the ACM - Vol. 37(8) - August '94.

11 Meta-discourse, as opposed to communication about the topic, is communication about the communication process.

[Eden,Ackermann'92] Eden, C. / Ackerman, F. "Strategy Development and Implementation - The Role of a Group Decision Support System"; in "Computer Augmented Teamwork: A Guided Tour" (ed. Bostrom, R. P.; Watson, R. T.; Kinney, S. T.); Van Nostrand Reingold, 1992.

[Kenis'95] Kenis, D. "Improving Group Decisions: Designing and Testing Techniques for Group Decision Support Systems Applying Delphi Principles"; PhD. at the University of Utrecht, 1995.

[KenisEtAl'95] Kenis, D. / Demeyer, S. / Maréshall, T. / Boyen, N. "Concluding Session using the DSide GDSS System"; EURO GDSS Workshop, Tilburg University, Netherlands; 20/21st April 1995.

[KenisEtAl'96] Kenis, D. / Lybaert, W. / Maréshall, T. "D-Side: a Worl-Wide Web Based GDSS"; EURO GDSS Workshop, Grenoble, France; 27/28 March 1995.

[Trahand'93] Trahand, J. "Structured Electronic Conversations"; Presentation at the European GDSS and Negotiation Workshop; Strathclyde Graduate Business School; 4/5th November 1993.

[Conklin,Begeman'88] Conklin, J. / Begeman, L. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion"; CSCW'88 Proceedings, ACM Press, 1988.

[NunamakerEtAl'91] Nunamaker, J.F. / Dennis. A. R. / Valachi, J. S. / Vogel, D. R. / George, J. F. "Electronic Meeting Systems to support Group Work"; Communications of the ACM, Vol. 34(7), July '91.