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 *informationrepositories* 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

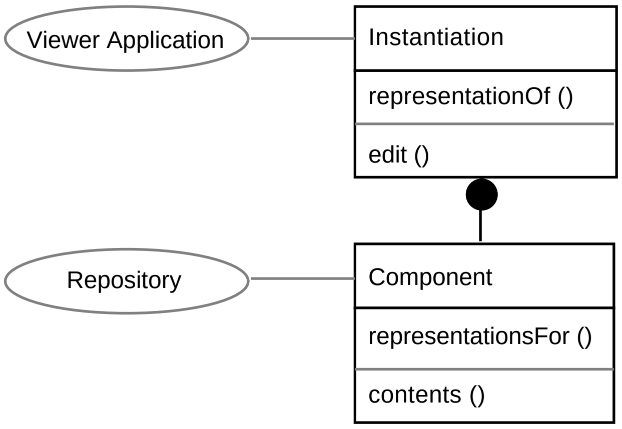
The Zypher framework browser incorporates among others the following viewer applications: a text editor, the Microsoft Word third party application, a HTML-browser, the standard Smalltalk code browsers (system, class, senders and implementors), three Zypher code browsers (class, senders and implementors)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

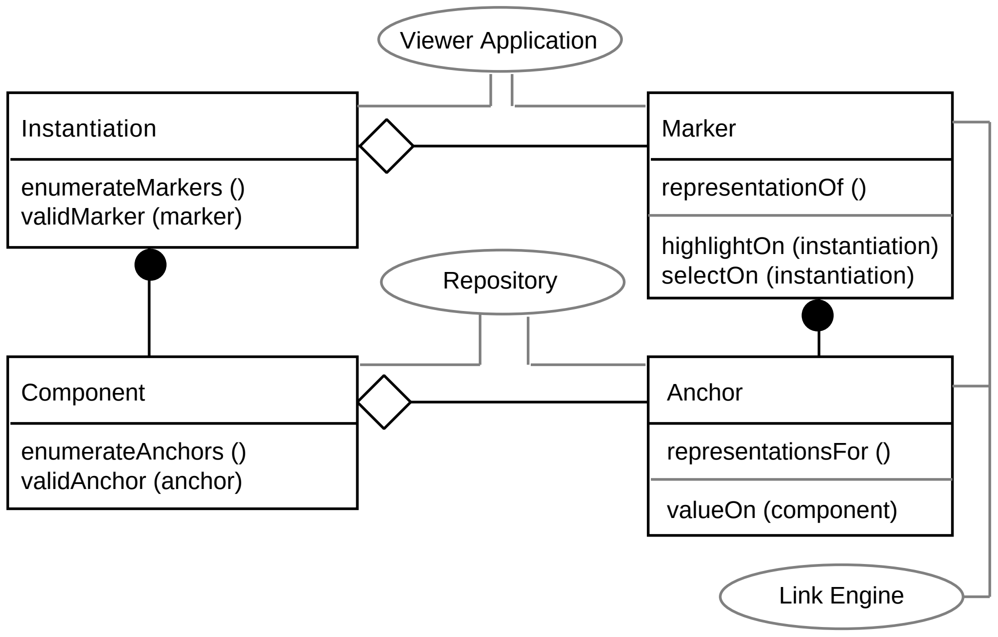
Translate the *unawareness constraint* in contracts stating that components / instantiations are not allowed to send messages to anchors / markers directly; that anchors are not allowed to send messages to markers or other anchors directly and that markers are not allowed to send messages to other markers 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].