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

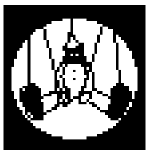
The notion of layering exists in hypermedia research as well, among others in the Hypertext Abstract Machine [Campbell,Goodman'87] and the Dexter model [Halasz,Schwartz'90]. There, layering releases the same tension between ease of use and extensibility, although there is a slight shift of focus as layers 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).

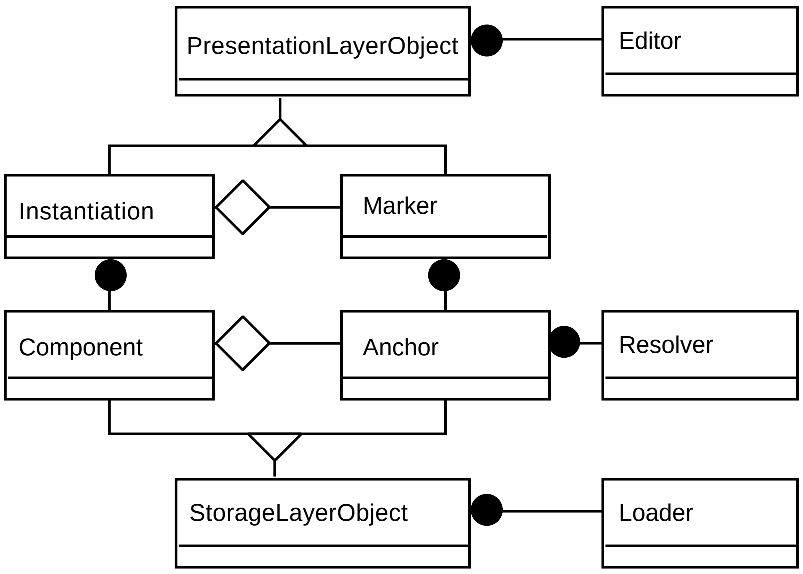


Figure 20: Overview of the Structure

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].

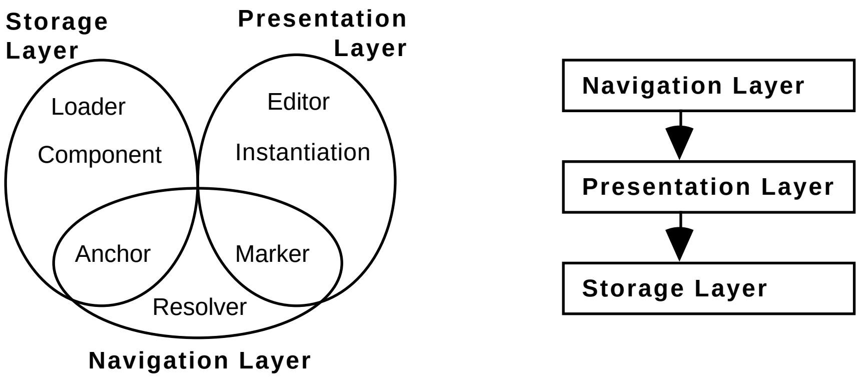


Figure 21: Layered Structure

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 *presentationlayer* 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

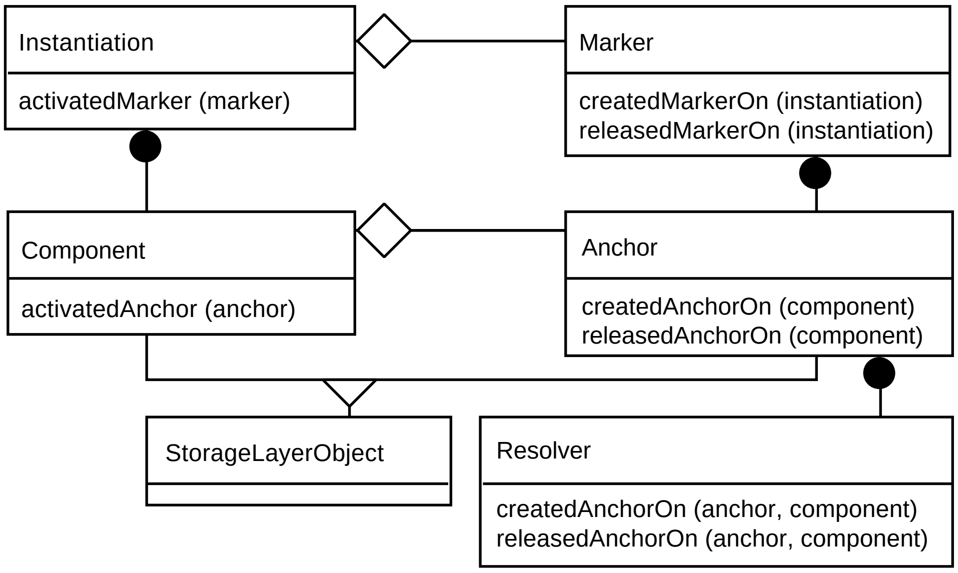
To synchronise the text editor and HTML-browser, both viewer applications should have an instantiation working on the same file component; the text editor is used to modify the contents in HTML-format, while the HTML-browser is used to inspect the layout of the document. When the text editor modifies 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


Instantiation
activatedMarker (marker)
Marker
createdMarkerOn (instantiation)
releasedMarkerOn (instantiation)
Component
activatedAnchor (anchor)
Anchor
createdAnchorOn (component)
releasedAnchorOn (component)
StorageLayerObject
Resolver
createdAnchorOn (anchor, component)
releasedAnchorOn (anchor, component)

(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.

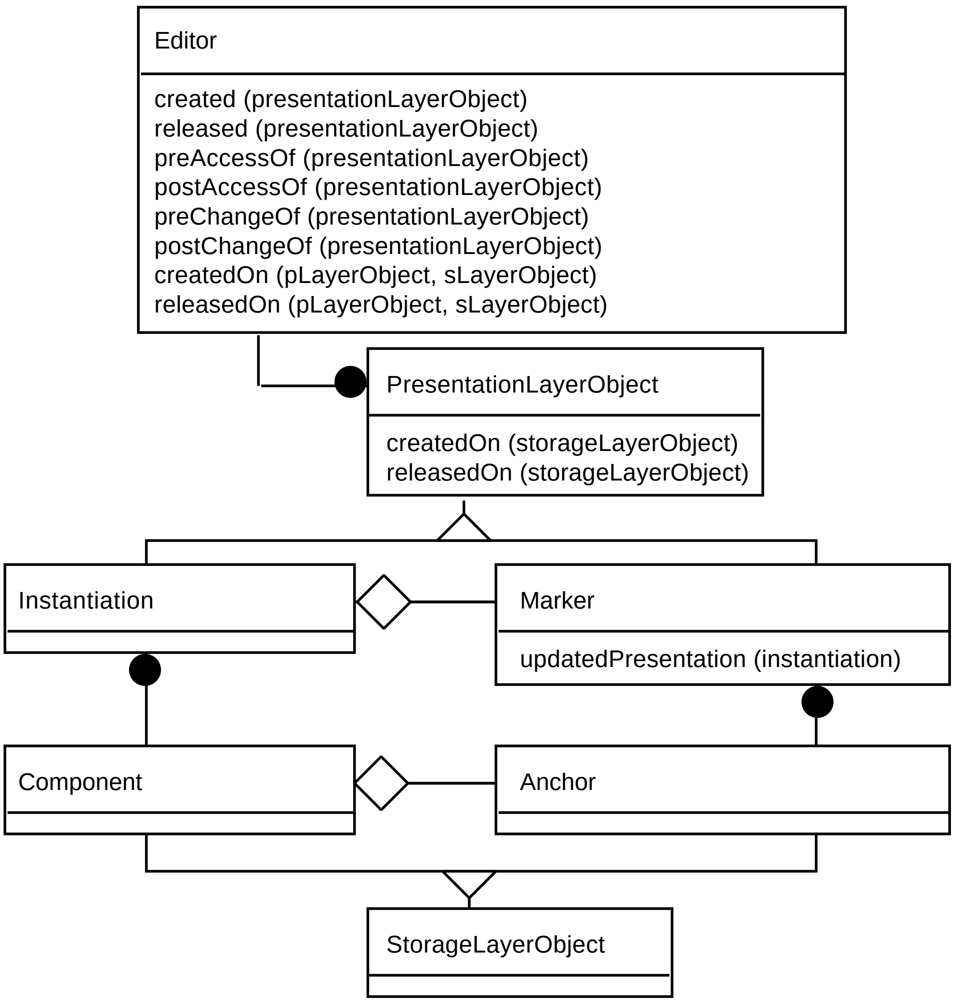
Anchor

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


Editor
created (presentationLayerObject)
released (presentationLayerObject)
preAccessOf (presentationLayerObject)
postAccessOf (presentationLayerObject)
preChangeOf (presentationLayerObject)
postChangeOf (presentationLayerObject)
createdOn (pLayerObject, sLayerObject)
releasedOn (pLayerObject, sLayerObject)
PresentationLayerObject
createdOn (storageLayerObject)
releasedOn (storageLayerObject)
Instantiation
Marker
updatedPresentation (instantiation)
Component
Anchor
StorageLayerObject

(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

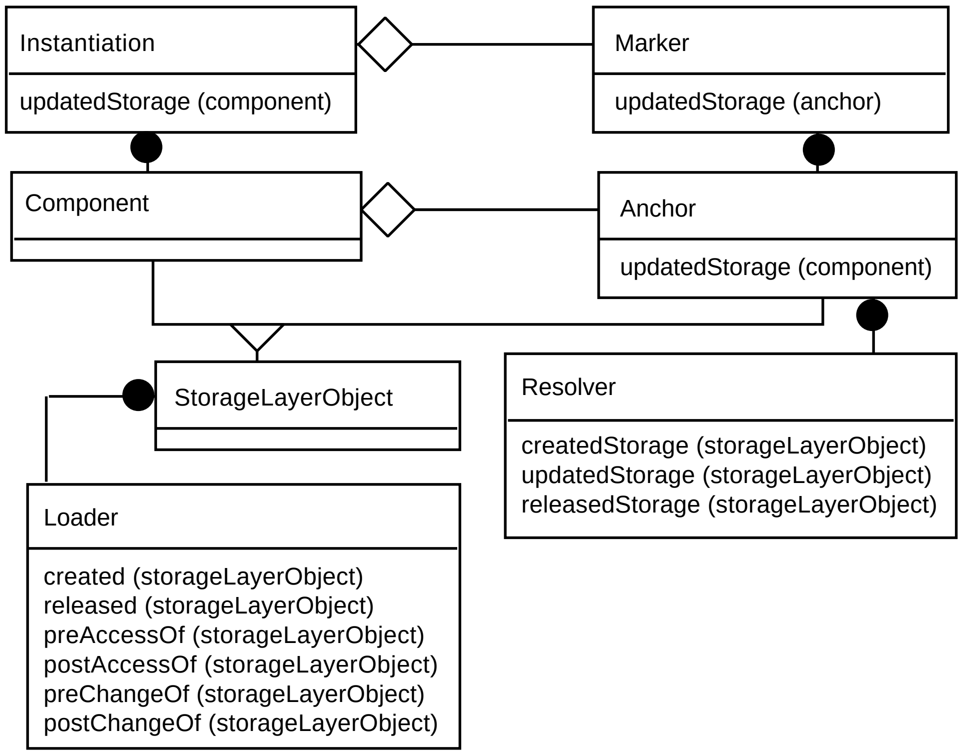
All markers implement an `updatedPresentation` message sent by whatever object that alters an instantiation associated with that marker. This allows the marker to adjust its internal state after the instantiation was altered. See [meta-objects ~ contracts] for further details.

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



Instantiation
updatedStorage (component)
Marker
updatedStorage (anchor)
Component
Anchor
updatedStorage (component)
StorageLayerObject
Resolver
createdStorage (storageLayerObject)
updatedStorage (storageLayerObject)
releasedStorage (storageLayerObject)
Loader
created (storageLayerObject)
released (storageLayerObject)
preAccessOf (storageLayerObject)
postAccessOf (storageLayerObject)
preChangeOf (storageLayerObject)
postChangeOf (storageLayerObject)

(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].