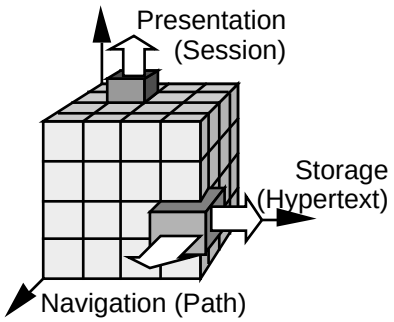
CHAPTER 7

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

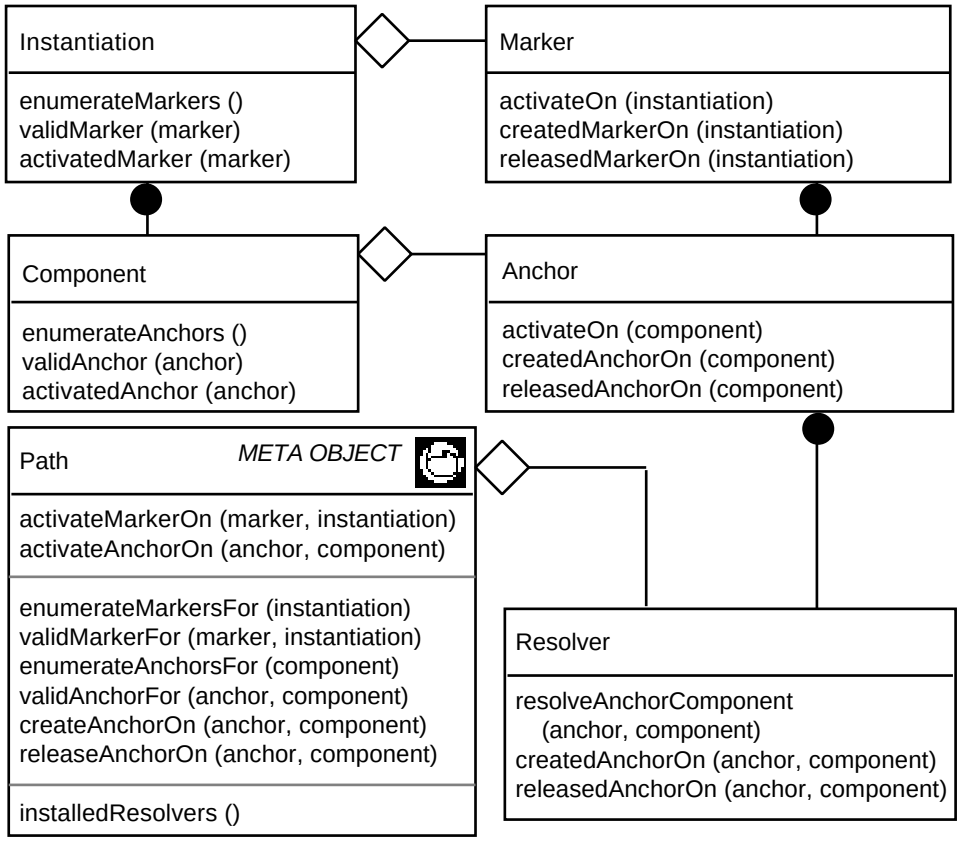
Domain level functionality is already furnished with the classes and interfaces specified in [resolver], [editor], [loader]. What is left implicit are the contracts between the different classes and that is precisely where the system behaviour is specified. So, to tailor the system behaviour one must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

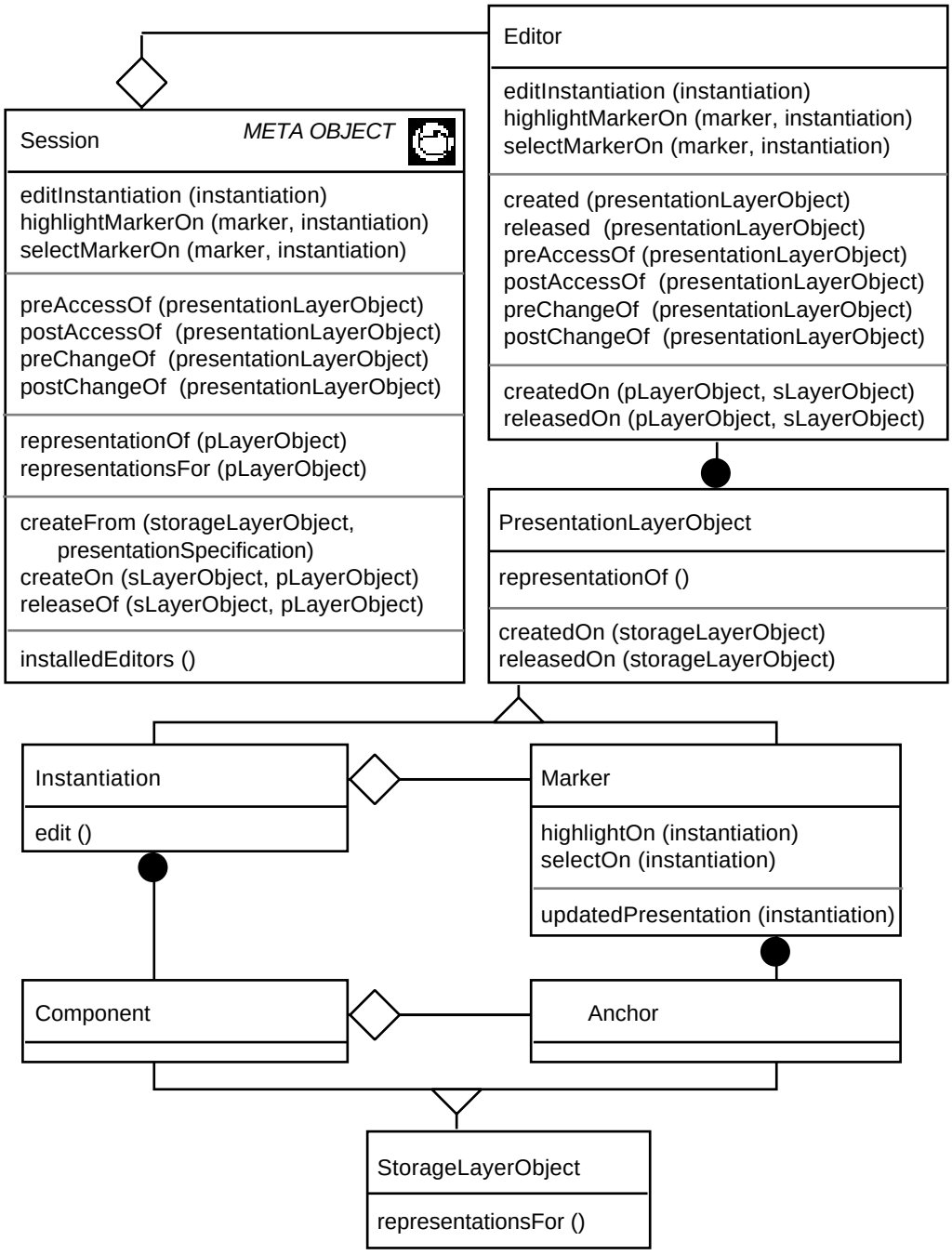
The Path is responsible for the maintenance of the aggregation relation between a component (instantiation) and an anchor (marker). For this purpose, the Path

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

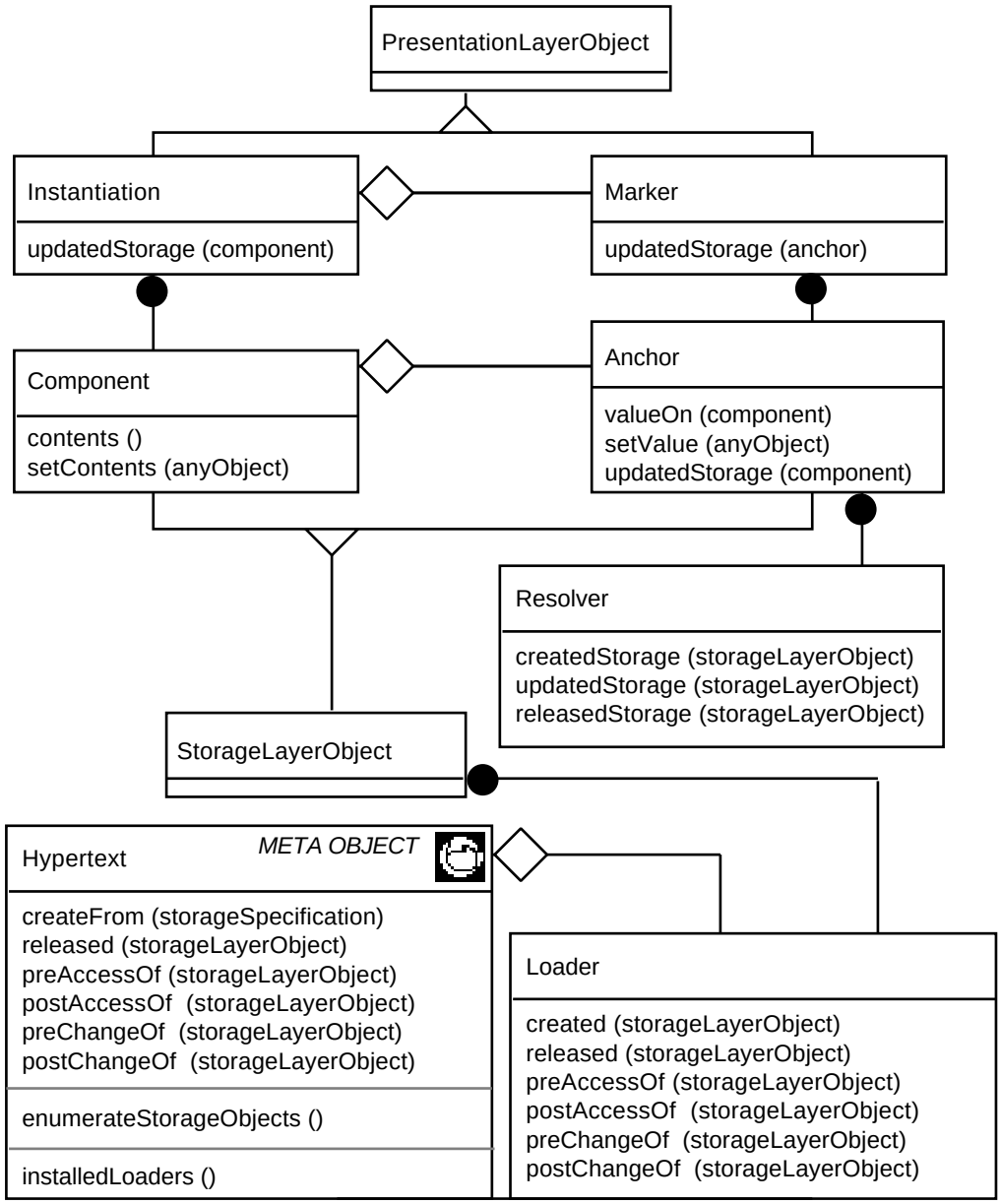
sent by any object that wants to alter the 1-to-many association between a component (anchor) and an instantiation (marker). The implementation must notify the instantiation (marker) and the editor by means of the `createdOn`, `releasedOn` messages.

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 '*causallyconnected*' 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].

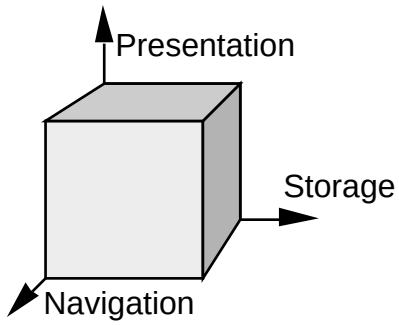


Figure 22: Zypher design space

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.

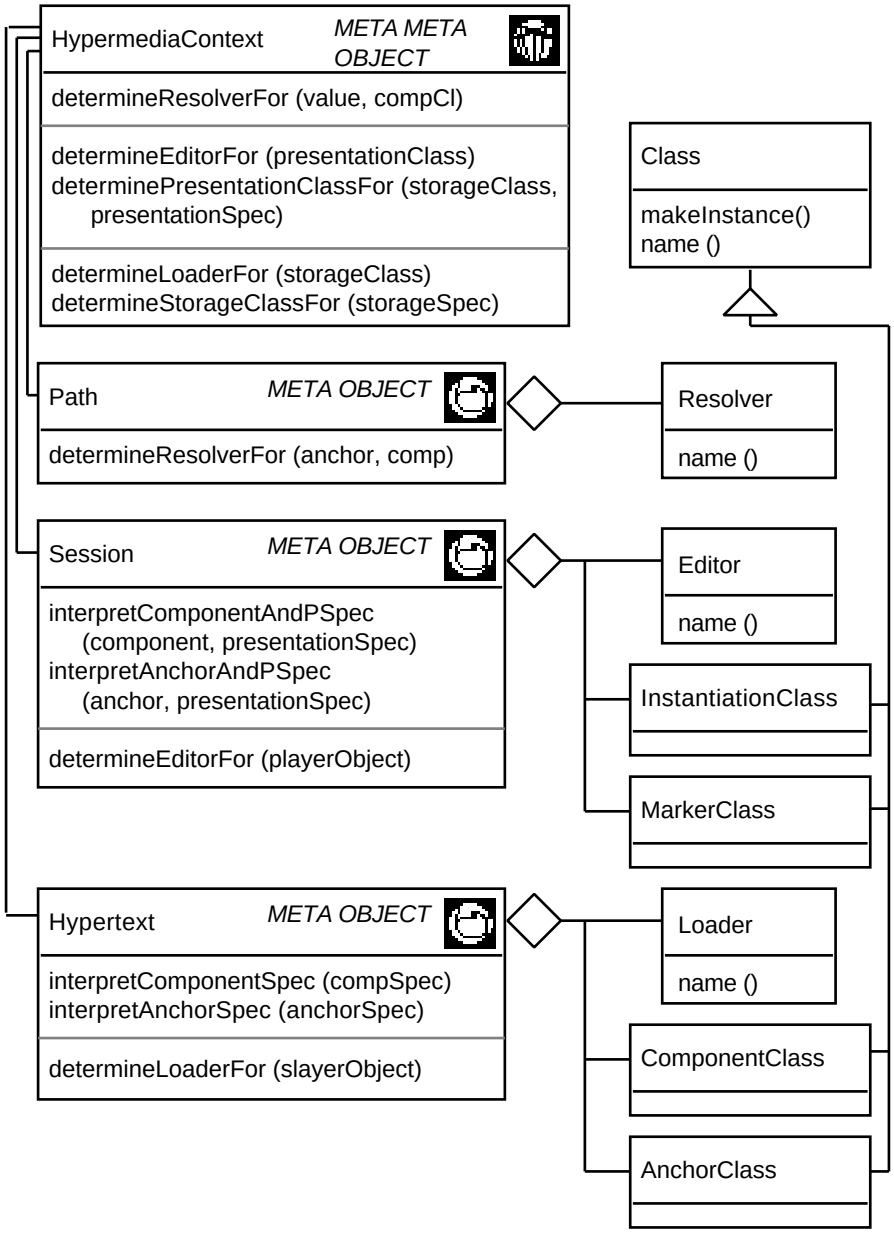
Also, the contracts on the meta-objects include operations to create, query and release associations between objects (StorageLayerObject - PresentationLayerObject in Session; Instantiation - Marker and Component - Anchor in Path) plus relations between objects and peripheral objects (PresentationLayerObject - 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.