Encapsulation and information hiding as the keys to maintainable and reusable hypermedia applications

Wilfried Lemahieu

Department of Applied Economic Sciences
Katholieke Universiteit Leuven
Naamsestraat 69 B-3000 Leuven
Belgium
tel: + 32 16 32 68 86
fax: + 32 16 32 67 32
e-mail: wilfried.lemahieu@econ.kuleuven.ac.be

1 Abstract

This paper presents a solution to the maintenance problem in hypermedia by applying object-oriented techniques to both the hypermedia data model and the hypermedia system's actual implementation. First, the primary concepts of the "*MESH*" *(Maintainable, End user friendly, Structured Hypermedia)* approach are discussed briefly. These consist of a *conceptual data model*, a *navigation paradigm* and an *implementation framework*. Thereafter, it is shown how the object-oriented concepts of *encapsulation* and *information hiding* result in a hypermedia system consisting of self-contained, independently coded nodes. *Intra node maintenance* is separated entirely from *inter node maintenance*: the hyperbase's link structure can be updated without affecting node content, whereas an individual node's multimedia content can be reorganized without necessitating updates to links or link anchors.

2 Introduction: object orientation and hypermedia

2.1 Hypermedia design based on a conceptual data model

Although the World Wide Web contributed tremendously to the popularity of the hypermedia paradigm, it also amply illustrated its two primary weaknesses: the problem of *user disorientation* and the difficulty of *maintaining* the hyperbase. The "lost in hyperspace" phenomenon is widely known in literature, e.g. [3]; [4]: whereas non-linear navigation is a very powerful concept in allowing the end user to choose his own strategy in discovering an information space, the resulting navigational freedom may easily lead to cognitive overhead and disorientation.

Equally stringent is the *maintenance* problem [23]. The latter was certainly less than a sinecure in the pioneering hypermedia implementations. A heavy burden upon hyperbase maintainability is the fact that, due to the absence of workable abstractions, many hypermedia systems implement links as direct references to the target node's *physical location* (e.g. the *URL* in a *WWW* environment). To make things worse, these references are embedded within the *content* of a link's source node [6]. As a result, moving a single node demands heavy maintenance to restore hyperbase integrity; *all* nodes' bodies have to be searched for a reference to the now-obsolete location and all found references have to be updated. Hyperbase maintenance has become a synonym for manually editing the nodes' *contents*. Whereas manually created links already reduce maintainability to a great extent, they also have a disastrous impact upon *consistency* and *completeness* [1]. The inability to enforce integrity constraints and submit the network structure to consistency and completeness checks, results in a hyperbase with plenty of *dangling links*. Needless to say that the consequences of inferior maintenance will also frustrate the end user and effect into additional orientation problems.

More recently, it has been suggested that abstractions such as node and link types offer increased consistency in both node layout and link structure with the added bonus of a navigational structure more comprehensible to the end user. The benefits of data modeling abstractions to both orientation and maintainability were already acknowledged in [12]. They yield richer domain knowledge specifications and more expressive querying. Typed nodes and links offer increased consistency in both node layout and link structure [16]; [26]. Higher-order information units and perceivable equivalencies (both on a conceptual and a layout level) greatly improve orientation [11]; [27]. Semantic constraints and consistency can be enforced [1]; [10], tool-based development is facilitated and reuse is encouraged [22].

Consequently, hypermedia design is to be based on a firm *conceptual data model*. The pioneering conceptual hypermedia modeling approaches such as *HDM* [9] and *RMM* [14] were based on the entity-relationship paradigm. Later on, object-oriented techniques were applied, both at the *conceptual* and the *implementation* levels. In some cases, object-orientation was primarily used in *hypermedia engines*, to model functional behavior of an application's *components*, e.g. *Microcosm* [5], *Hyperform* [29] and *Hyperstorm* [2]. Other approaches, such as the *Tower model* [7], *EORM* [17] and *OOHDM* [24], modeled the *application domain* by means of the object-oriented paradigm.

2.2 The MESH hypermedia framework

This paper introduces *MESH* (*Maintainable, End user friendly, Structured Hypermedia*), which combines an *object-oriented modeling* approach with a fully object-oriented *implementation* [18]. Based on the conceptual modeling abstractions, it offers a *context-based navigation paradigm* to accommodate for user orientation.

*MESH'*s data model builds on concepts and experiences in the related field of database modeling, taking into account the particularities inherent to the hypermedia

approach to data storage and retrieval. Established object-oriented modeling abstractions [15]; [25] are coupled to proprietary concepts to provide for a *formal hypermedia data model*. While uniform layout and link typing specifications are attributed and inherited in a *static* node typing hierarchy, both nodes and links can be submitted *dynamically* to multiple complementary classifications. The data model provides for a firm hyperbase structure and an abundance of meta-information that facilitates implementation of the enhanced navigation paradigm.

An elaborate description of the data model and the navigation paradigm can be found in [20] and [19] respectively. In both publications, the most important object-oriented concept is *abstraction*. Abstractions used in the conceptual model and the navigation paradigm facilitate both orientation and maintenance. By means of inheritance, node properties can be defined on a high level of abstraction, and be inherited and refined in more specific "*node types*", greatly reducing design and maintenance efforts. The same abstractions allow for the navigation paradigm to take account of the so-called *navigation context*. Guided tours are generated automatically along nodes relevant within this context, to "guide" the user and avoid disorientation.

However, object-orientation entails more than merely subtyping and inheritance. Another object-oriented concept that is applied successfully in *MESH* is *encapsulation*. This paper very briefly discusses the conceptual hypermedia data model and navigation paradigm, but its main focus is upon how encapsulation further facilitates design and maintenance. At the *conceptual* level, it allows for nodes to be considered as independent entities, which can be developed in parallel by different parties. Any node can be designed without the need for knowing the entire hypermedia structure. At the *implementation* level, it allows for nodes to be considered as objects with a very loose coupling. They interact by means of a well-defined interface: their set of attributed link types, but can stay unaware of one another's actual implementation. As a consequence, each node can be updated without affecting the rest of the hyperbase, which obviously reduces the maintenance problem to a great extent. Each update, both in terms of *link structure* and of *node content* becomes a *local* operation, instead of a global affair with escalating side effects. Nevertheless, it will allow a very high degree of freedom regarding how a node's content is actually implemented, as long as the external view of a node corresponds to the link type based interface.

3 An overview of the MESH framework

3.1 The basic concepts: node types, layout templates and link types

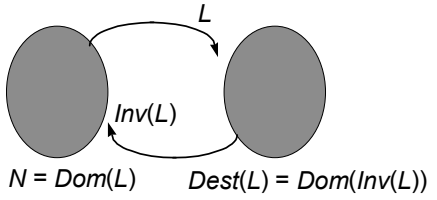
As with any hypermedia model (except for set-based paradigms), *MESH*'s basic building blocks are *nodes* and *links*. However, in *MESH*, these concepts explicitly take on the semantics of *objects* and *relationships* as in an object-oriented conceptual data model. On a conceptual level, a node can be considered as a *black box*, which communicates with the outside world by means of its links.

MESH's data model does not explicitly define the notion of *anchors*. A link always refers to a node *as a whole*. True to the object-oriented *information-hiding* concept, no direct calls can be made to a node's *properties*, i.e. its multimedia content. However, internally, a node may encode the intelligence to adapt its visualization to the *navigation context*, as discussed in a later section.

Nodes are assorted in an inheritance hierarchy of *node types*. Each child node type should be compliant with its parent's definition, but may fine-tune inherited features and add new ones. These features comprise two concepts: node *layout* and node *interrelations*, abstracted in *layout templates* and *link types* respectively. Whereas link types are well-defined at the conceptual level, a node's layout template will depend upon the actual implementation environment, e.g. as to the Web it may be HTML or XML based. As *MESH* separates the inter-node data modeling aspect from intra-node design, this section's discussion regarding inheritance mainly concerns the inheritance of link types. With regard to node layout, we will suffice by stating that with any level in the node typing hierarchy, a template can be associated, where each template is a refinement of its immediate ancestor. The multimedia objects of a node type's instances are to comply with the corresponding layout template. Node typing as a basis for layout design allows for uniform behavior and onscreen appearance for nodes representing similar real world objects.

A *link* represents a one-to-one association between two nodes, with both a semantic and a navigational connotation. A link is always *directed* and offers an access path from its *source* to its *destination node*. Directionality is important for two reasons: first there is a *semantic* aspect, because the exact meaning of a relation might otherwise be confusing, e.g. for the relation ***is-a-parent-of***. Second, because of the *navigational* aspect, where a source and a destination are inherent to each navigation step.

Links representing similar semantic relationships are assembled into *types*. Link types are attributed to node types and can be inherited and refined throughout the hierarchy. In *MESH*, definition of a link (type) automatically effects into the definition of an inverse link (type). Only the source node of a link is made explicit, the destination is defined as the source of its inverse. So if a link is added to a node, the destination node must belong to the domain of the inverse link type: a node of type *N* can be linked by a link of type *L* to any node that belongs to the domain of *L*'s inverse. A link type's *destination* is a derived property, defined as the *inverse link type's domain*.



Link type properties such as *domain*, *cardinalities* and *destination/inverse* allow for enforcing constraints on their instances. These properties can be overridden to provide for stronger restrictions upon inheritance. E.g. whereas an **artist** node can be linked to

any **artwork** through a ***has-made*** link type, an instance of the child node type **painter** can only be linked to a **painting**, by means of the more specific child link type ***has-painted***.

3.2 The use of aspects to overcome limitations of a rigid node typing structure

3.2.1 Definition of aspect descriptor and aspect type

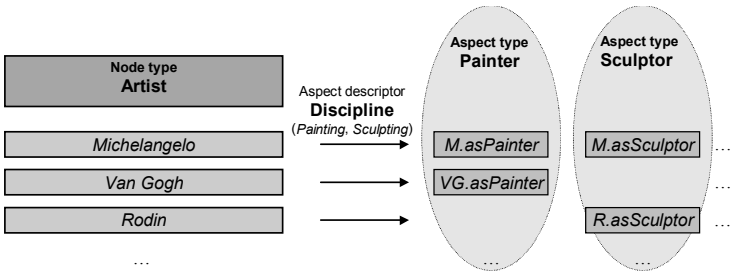
As the above model will also be the basis for node layout design, we deliberately opted for a single inheritance structure, where node classification is total, disjoint and constant (see [18] for a more thorough discussion). However, aspects can provide an elegant solution in many situations that would otherwise call for multiple inheritance. The aspect construct allows for defining *additional* classification criteria, which are not necessarily subject to the restrictions of being total, disjoint and constant. Apart from a single "most specific node type", they allow a node to take part in other secondary classifications that are allowed to change over time.

An *aspect descriptor* is defined as an attribute whose (discrete) values classify nodes of a given type into respective additional subclasses. In contrast to a node's "main" subtyping criterion, such aspect descriptor should not necessarily be *single-valued* nor *constant over time*. Aspect descriptor properties denote whether the classification is *optional*/*mandatory*, *overlapping*/*disjoint* and *temporary*/*permanent.*

Each *aspect type* is associated with a single value of an aspect descriptor. An aspect type defines the properties that are attributed to the class of nodes that carry the corresponding aspect descriptor value. An aspect type's instances, *aspects*, implement these type-level specifications. Each aspect is inextricably associated with a single node, adding characteristics that describe a specific "aspect" of that node.

A node instance may carry multiple aspects and can be described by as many aspect descriptors as there are additional classifications for its node type. If multiple classifications exist, each aspect descriptor has as many values as there are subclasses to the corresponding specialization. Its cardinalities determine whether the classification is total and/or disjoint. As opposed to node types, aspects are allowed to be volatile. Hence, dynamic classification can be accomplished by manipulating aspect descriptor values, thus adding or removing aspects at run-time. Aspect types attribute the same properties as nodes: *link types* and *layout*. However, their instances differ from nodes in that they are not directly referable. An aspect represents the *same real-world object* as its associated node and can only be visualized as a subordinate of the latter.

E.g. to model an **artist** that can be skilled in multiple disciplines, a non-disjoint aspect descriptor *discipline* defines the **painter** and **sculptor** aspect types. Discipline-specific node properties are modeled in these aspect types, such that e.g. the **Michelangelo** node features the combined properties of its **Michelangelo.asPainter** and **Michelangelo.asSculptor** aspects.



3.2.2 Aspect types as node type building blocks

Node type properties (i.e. layout and link types) can be *delegated* to aspect descriptors, such that they can be inherited and overridden in each aspect type that is associated with one of the descriptor's values. An aspect type's *layout* template refines layout properties that are delegated to the corresponding aspect descriptor. Link types delegated to an aspect descriptor can be inherited and overridden as well. In addition, each aspect type can define its own supplementary link types. The inheritance/overriding mechanism is similar to the mechanism for supertypes/subtypes, but because an aspect descriptor can be multi-valued, particular care was taken so as to preclude any inconsistencies (see [18] for further details).

Aspect types themselves are node type properties that can be inherited and overridden across the node type hierarchy. The *aspect descriptor* is used as a vehicle for the inheritance of aspect types. This ability yields the opportunity to use aspects as real building blocks for nodes. Link types and layout definitions pertaining to a single "role" a node may have to play, can now be encapsulated into one aspect type. If the corresponding aspect descriptor is attributed at a generic level in the node hierarchy, the aspect type can be inherited where necessary by more specific node types. This allows for the modeling of a similar 'aspect' in otherwise completely unrelated node types. Node types can be 'assembled' by inheriting the proper aspect types, complemented by their own particular features. In this way, different aspects associated with the same node instance can have different editing privileges, such that updating multimedia *content* can be delegated to different parties.

3.3 Link typing and subtyping

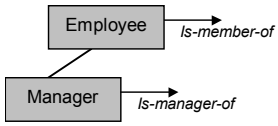
In common data modeling literature, subtyping is invariably applied to *objects*, never to *object interrelations*. If additional classification of a relationship type is called for, it is *instantiated* to become an object type, which can of course be the subject of specialization. However, as for a hypermedia environment, node types and link types are two separate components of the data model with very different purposes. It would not be useful to instantiate a link type into a node type, since such nodes would have *no content* to go along with them and thus each instance would become an 'empty' stop during navigation.

This section demonstrates how specialization semantics can be enforced not only upon node types, but also upon the *link types*. A sub link type will model a type whose set of instances constitutes a subset of its parent's, and which models a relation that is more specific than the one modeled by the parent. Link types are deemed extremely important, as they not only enforce semantic constraints but also *interface* between nodes, such that these can be coded and updated independently of one another. Moreover, they provide the basis for *context-sensitive node visualization*, as discussed further on in this paper.

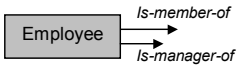
A link instance is defined as a source node - destination node tuple (n_s, n_d). Tuples for which this association represents a similar semantic meaning are grouped into link types. A link type defines instances that comply with the properties of the type and is constrained by its *domain*, its *cardinalities* and its *inverse link type*. The *domain* of the link type is the data type to which the link type is attributed. This can be either a node type or an aspect type.

If L_c is a sub link type resulting from a specialization over L_p, the set of (n_s, n_d) tuples defined by L_c is a subset of the one defined by L_p. Such specialization is called *vertical* if it is the consequence of a parallel classification over the link types' *domain*, denoting that the sub link type is attributed at a 'lower', more specific level in the node typing hierarchy than its parent. If L_c and L_p share the same domain, L_c can still define a subtype of L_p in the case where L_c models a more restricted, more specific kind of relationship than L_p, independently of any node specialization. Both parent and child link type are attributed at the same level in the node type hierarchy, hence the term *horizontal* specialization.

E.g.



***Vertical* link specialization**



***Horizontal* link specialization**

Apart from the domain, a link type's cardinalities and inverse can be overridden as well upon specialization. The *cardinalities* determine the minimum and maximum number of link instances allowed for a given source node. *MESH* presents a formal overriding mechanism, wherein particular care is taken so as not to violate the parent's constraints, particularly in case of a non-disjoint classification. For further details we refer to [18].

3.4 MESH's context-based navigation paradigm

The navigation paradigm as presented in *MESH* combines set-based navigation principles with the advantages of typed links and a structured data model. The typed links allow for a generalization of the *guided tour* construct. The latter is defined as a linear structure that eases the burden placed on the reader, hence reducing disorientation [28].

As opposed to conventional static guided tour implementations, *MESH* allows for complex structures of nested tours among related nodes to be generated *at run-time*, depending on the *context* of a user's navigation. Such context is derived from *abstract navigational actions*, defined as link type selections. Indeed, instead of selecting a single *link instance*, similarly to the practice in conventional hypermedia, a navigational action may also consist of selecting *an entire link type*. Selection of a *unique link type* results in a single destination node being accessed, e.g.: **Sunflowers.*exhibited-in*** := **National Gallery**.

Selection of a non-unique link type from a given source node results in a *guided tour along a* set of nodes being generated. This tour includes all nodes that are linked to the given node by the selected link type, e.g.: **Van Gogh**.***has-painted*** := {**Potato eaters**, **Self portrait**, **Sunflowers**,…}.

Navigation is defined in two orthogonal dimensions: on the one hand, navigation *within the current tour* yields linear access to complex webs of nodes related to the user's current focus of interest. On the other hand, navigation *orthogonal to a current guided tour*, changing the context of the user's information requirements, offers the navigational freedom that is the trademark of hypertext systems. In addition, the abstract navigational actions and tour definitions sustain the generation of very compact overviews and maps of complete navigation sessions. This information can also be *bookmarked*, i.e. bookmarks not just refer to a single node but to a complete navigational situation, which can be resumed at a later date.

Important to this paper, is that each navigational action can be described in terms of a *link type*: navigation *within* the current tour is defined by the link type that defines the tour's context. Navigation orthogonal to the current tour can be described by the newly selected (unique or non-unique) link type.

3.5 A platform-independent implementation framework

3.5.1 Separation of navigation structure from node layout

The navigational paradigm presented in the previous section requires a hyperbase that is *searchable* for its link structure: to generate the necessary guided tour links at run-time, the application needs to be able to query the hyperbase for nodes related to the current context. As a consequence, there are two alternatives for hyperbase implementation. The first one is to encapsulate all links within the body of the nodes, like it is the case in many hypermedia environments, such as standard HTML pages in the WWW. However, unlike many other environments, *MESH* should allow for all nodes to be queried for their link information. This would call for an object-oriented database system where each node is an object and where all links are represented as symbolic pointers to other objects, which can be queried by means of an object-oriented query language.

However, forcing all nodes with their (possibly very distinct data formats) into one proprietary object-oriented database model would result in an unacceptable lack of

openness and dependence upon one specific object-oriented DBMS. Therefore, a second alternative was opted for, where the *information content* and *navigation structure* of the nodes are separated and stored distinctly into storage devices that are tailored to the specific needs of the type of information stored. A simple relational database can be used to capture the link structure and meta-information of the hypermedia system, along with references to the physical addresses of the corresponding nodes. This option leaves much more freedom to implement the *content* of a node.

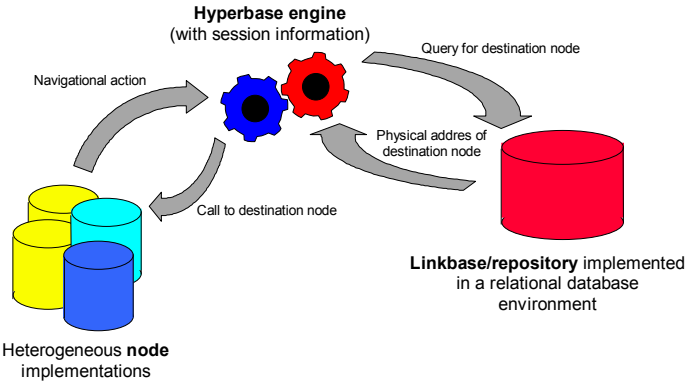
3.5.2 MESH's implementation architecture

The resulting system consists of three types of components: the *nodes*, the *linkbase/repository* and the *hyperbase engine*. In [18], the implementation framework was deliberately kept independent of any actual software platform. However, the current prototype is Web-based.

The *nodes* side of the hypermedia system is considered as a potentially *heterogeneous* collection of entities, ranging from flat files (e.g. HTML fragments) to objects in an object-oriented database, each containing one or more embedded multimedia objects. Nodes are very loosely specified. They only have to be associated with a filename or any other *unique identification* and should be able to return a *navigational action* (see below) upon closure. However, since link information is stored separately of the nodes, a node does not have to be a searchable object. Its internal content is shielded from the outside world by the indirection of link types playing the role of a node's *interface*.

Linkage information is not embedded in a node's body. Instead, links as well as meta data about node types, link types, aspect descriptors and aspects are captured within a searchable *linkbase/repository* to provide the necessary information pertaining to the underlying hypermedia model, both at design time and at run-time. This repository is implemented in a relational database environment. Only here, references to physical node addresses are stored, these are never to be embedded in a node's body. All external references are to be made through location independent node *ID*'s.

The *hyperbase engine* is conceived as a server-side application (the current prototype is servlet-based) that listens for navigational actions issued from the current node, retrieves the correct destination node, keeps track of session information and provides facilities for generating maps and overviews. Since all relevant linkage and meta information is stored in the relational DBMS, the hyperbase engine can access this information by means of simple, pre-defined and parameterized *database queries*, i.e. without the need for searching through *node content*.



As described above, nodes do not refer directly to one another. Rather, node interaction is based on their *attributed link types* and mediated by the *hyperbase engine*. The interaction mechanism can be compared to object-oriented *method calls* and *return values*, with the link types defining a node's *interface*. The implementation of these methods is embedded in a node's body and shielded from the outside world, according to the object-oriented *encapsulation* and *information hiding* principles. The remainder of this paper discusses this interaction mechanism and indicates how it greatly facilitates hyperbase development and maintenance.

4 An object-oriented approach to node interaction

4.1 The anchor notion

The traditional concept of an *anchor*, as defined e.g. in [13], is purposed at allowing a link to be associated with an *internal component* of a node. In this respect, its applicability is twofold: on the one hand it allows for an *incoming* link to refer directly to one or more of a node's embedded multimedia objects. On the other hand, it allows for an *outgoing* link to be selected from the node component it is anchored to, e.g. by clicking the anchor.

If the granularity of linking is to be more delicate than simply connecting *entire nodes*, some sort of anchoring is indispensable. Several hypermedia approaches, such as [7], [13] and [21], consider anchors as first-class objects, i.e. an anchor is a full-fledged hypermedia component. Links are defined between two (or more) anchors, rather than between nodes. Having anchors as separate constructs, independent of the links, certainly has the advantage that the linking mechanism is not burdened by the "internal node affair" of anchoring the link within the node content. This is especially important if the node content may consist of heterogeneous media types, possibly requiring completely different methods of anchoring (e.g. movie sequences versus textual media).

From a pure data modeling point of view, it is not necessary to discriminate *source anchors* from *destination anchors* [7]; [13]. Both have the same purpose: referring to internal components of a node's content. However, on a behavioral level, there certainly is a difference [8]; [17]. A source anchor is to induce a navigational action

upon stimulation, hence it should be able to receive some kind of user input. The most well-known example is the traditional button or underlined word. A destination anchor influences the *visualization* of a node and may work upon one or more multimedia objects in a node's content. It is narrowly coupled to a node's *presentation methods*. E.g. a source anchor to a link between a **painting** and its **painter** should be able to provoke a navigational step from the **painting** to the **painter**, whereas a destination anchor (to the same node) should determine how the **painting** instance is to be visualized, given it is accessed through the corresponding link.

4.2 Encapsulation versus anchoring

Both source and destination anchors have the property of "pointing" to one or more specific multimedia objects *within* a given node. Consequently, if a node is seen as an object, the anchor concept violates the *encapsulation* and *information hiding* principles of object-orientation. These principles state that an object is to encapsulate all functionality necessary to manipulate its own state. It should hide its properties and method implementations from the outside world and is to offer only a limited *interface* for external objects to call upon. Through this interface, the external objects communicate with the object and use its services, ask for embedded information etc. External objects should not have knowledge of an object's internal properties. This principle is very advantageous in terms of maintainability and reuse: the internal features of the object can be changed drastically without affecting other objects, as long as the interface to the outside world remains unchanged. An object can even be replaced by a different object, as long as a similar interface is offered.

An anchor object, be it a source or destination anchor, violates the information hiding concept by referring to a node object's internal (multimedia) components. To benefit from the information hiding principle, a source and destination anchor should be known only to a link's respective source and destination nodes. Therefore, *MESH* does not define real anchor components that can be referenced externally, but leaves anchoring to the *internal node design* instead. Links are directly defined between *nodes*, not between *anchors*. Both a node's "incoming" and "outgoing" links are dealt with *internally*, by the node itself.

4.3 An object-oriented alternative to anchoring: link types as the interfaces for node interaction

Indeed, instead of explicit anchoring, *MESH* uses a node's *attributed link types t*o interface between the global hyperbase objects and the node's internal components.

The node provides the user with a user interface (defined in the layout template) to interact with the node and explore its embedded multimedia objects. The association of a multimedia object with a navigational action can be seen as the equivalent of a *source anchor*: if the multimedia object is suitably 'stimulated' by the user, the corresponding user interface event causes a *navigation step*. As a consequence, the

current node is *closed*, i.e. it is abandoned in favor of another node to be *accessed* and to become the 'new' current node. Upon closure, the node passes a return value to the hyperbase engine. This return value depends on the event that induced the navigation step and provides the hyperbase engine with a means of calculating the appropriate *destination node* as the next/previous node in a guided tour, the single destination node of a unique link type or the first node in a newly started guided tour, defined by a non-unique link type.

Because of *MESH*'s navigation mechanism, the value returned to the hyperbase engine by the closing node will actually be a link type ID. The link type not only determines *which node* will be accessed next, but also *which visualization method* will be called upon this destination node. The *destination anchor* concept is generalized by the so-called *context sensitive visualization* principle: a node's visualization is made sensitive to the *context*, defined by the *link type*, within which it is accessed. Each link type corresponds to a *presentation routine*, which provokes a befitting visualization of the node's multimedia objects within a particular context. Hence the same node will visualize itself differently, depending on the context in which it is accessed. The latter is accomplished without a link referring to the actual multimedia objects: the appropriate behavior is encapsulated and hidden within the node as a presentation routine's implementation.

Summarizing, node interaction is regarded as interaction between self contained *objects*. Each node/object defines its own routines for visualization and interaction with the user. When a user selects a navigational action in the current node, the latter closes and passes a *return value* to the hyperbase engine. The engine calculates the correct destination node from this return value and calls a *presentation method* upon the latter. The node's actual implementation is hidden; its presentation routines define its *public interface*. By associating a presentation routine with each link type attributed to a node type, the node type's instances are equipped with an appropriate visualization routine for each context in which they can be accessed. The two subsequent sections further discuss *MESH*'s alternatives to source anchors and destination anchors respectively.

4.4 Link type selections instead of source anchors

It was already discussed how navigational actions, both within the current tour and orthogonal to the current tour, can always be described by a *link type*. The latter defines the context within which the action takes place, or the new context induced by the action. Exactly such link type will make out the return value a node passes to the hyperbase engine.

Each node defines its own user interface, as specified in its type's template. Consequently, a user interacts with only a single node at a given time. It is this *current node*'s duty to accept the user's choice for the next navigational step and to present the hyperbase engine with an indication about which node to access next. How a user's choice for a navigation step is to be made known to a node, is left to its internal design. Like in any hypermedia environment, this can be accomplished

through clicking underlined words, hot spots, buttons, clickable maps etc. However, independently of the implementation, *MESH* defines a *source anchor* as the association between a *user interface event* and a *link type*. It causes the current node to *close* and pass the link type as a return value to the hyperbase engine. In contrast to other approaches, the anchor is not to be known outside the node: it is considered an internal node property and does not belong in the conceptual hypermedia model. Its implementation can vary from node to node, depending on the node's implementation and the corresponding multimedia object that induces the event.

MESH greatly improves node independence and maintainability by anchoring *link types* instead of *link instances* wherever possible. A link type anchor is independent of the node instance and can be defined once at an aggregate level in a node type's (or aspect type's) *layout template*. The "anchors" remain the same for each node (or aspect) instance, independently of the corresponding link instance(s). Whenever a new instance is defined, such anchor can be generated automatically. Upon stimulation of the anchor, the corresponding link type ID is passed to the hyperbase engine. Only here, it is mapped to one or more *link instances*. A unique link type is mapped to a unique destination node. A non-unique link type is mapped to a *guided tour*, of which the first participating node is accessed. Such guided tour is derived at runtime and consists of all destination nodes of link instances of the selected type, which have the current node as source node. They can be visited sequentially by the user. E.g. a source "anchor" to the link type ***has-painted*** can be defined in the layout template associated with the node type **painter**. At runtime, stimulating this anchor in any **painter** instance will provoke a guided tour along all paintings painted by this particular painter: **Van Gogh**.***has-painted*** := {**Potato eaters**, **Self portrait**, **Sunflowers**,…}.

Hence maintenance of the individual link instances does not affect the node's internal properties. As to non-unique link types, links can be added or removed without affecting the anchor and, consequently, the node's content. The correct guided tour is calculated at runtime by the hyperbase engine. This does not only facilitate development to a great extent, but also improves the user's grasp upon the underlying hypermedia structure by providing similar anchors to similar links. As such, cognitive overhead and the risk of disorientation are reduced.

In addition, since all relevant linkage and meta information is stored in a relational database, the hyperbase engine itself is always able to generate a separate *navigation panel* upon user request. This panel can provide the user with a complete *node overview*: a hierarchical index of all accessible destination nodes, based on the link typing hierarchy. Moreover, it could provide information about possible guided tours, local maps, fish-eye views etc. It is important to note that such information can be provided through simple, pre-defined and parameterized *database queries*, i.e. without the need for searching through *node content*. In addition, such navigation panel can provide an interface for user interaction in the case where the node collection includes "third-party" objects such as word processor or spreadsheet documents, which may not encompass a means for anchoring links themselves.

Finally, the navigation panel would also inform the user about what is called *non-advertised links* in [1], i.e. links that are not explicitly anchored. Indeed, most nodes will have many more links than the ones that are explicitly associated with one or more user interface events. This is partially a consequence of inverse links being automatically generated for each link added. E.g. whereas each **painting** may anchor a link to its **painter**, it may not be desirable for a **painter** to anchor links to each individual **painting**, although these links will be present in the linkbase. Rather will the link type ***has-painted*** be anchored, to start a guided tour of all of a painter's work. It's the designer's responsibility to decide which links will be anchored, weighing off the supply of additional information against the risk of cognitive overhead. However, this decision never affects navigational freedom, as all non-anchored links can be made visible by a system generated node overview.

4.5 Context sensitive visualization instead of destination anchors

4.5.1 A node type's layout template

Because a destination anchor referring to a node's internal multimedia objects violates the encapsulation and information hiding paradigm, *MESH* provides an alternative approach. Instead, a node can be endowed with the intelligence to tune its visualization to the *context* in which it is accessed.

Node visualization in *MESH* builds upon two elements: layout templates and presentation routines. The *layout template* associated with each *node type* and *aspect type* describes its instances' multimedia objects on an abstract level and enforces a uniform user interface and consistent node layout. The *presentation routine* associated with each *link type*, denotes how a node is to be visualized, when accessed through this link type, i.e. in a particular context. This section deals with the more general aspects of layout templates and node visualization. The section hereafter elaborates on the context-sensitive node visualization mechanism.

Indeed, as discussed in detail in [18], each node type is to be associated with its own layout template, such that all of its instances share a similar "look and feel". The template describes on an abstract level what multimedia objects should be available and defines a complete presentation framework of all information content encapsulated within a node instance. Designing a template for node presentation can be seen as attributing a set of *placeholders* towards the output device(s) and specifying how the respective placeholders should be filled up by a given node instance. E.g. a template could be defined as an XML DTD, combined with a style sheet. However, the notion of placeholders should be looked upon in a most general meaning, again depending on the possible media types. If the application includes audio, the audio track can also be seen as a placeholder. In the case where time dependent media play a critical role, the two spatial co-ordinates can be extended with an additional temporal co-ordinate. This framework is unique for a given node type and is independent of the link types through which node instances will be accessed.

Just like link types, layout templates can be inherited and overridden in both child node types and aspect types. Purposefully, the description of the layout inheritance and overriding mechanism is kept very general and abstract in [18]. The concrete approach will again depend on implementation environment, multimedia data types etc. No matter how, a consistent layout can be enforced across all node types, by defining common layout properties in an abstract level's template and *inheriting* and *refining* them at more concrete levels in the node typing hierarchy.

Associating *aspect types* with a layout template too allows for similar layout properties to be modeled orthogonally to the node type inheritance hierarchy. An aspect instance presents its own embedded multimedia data, as determined by the aspect type's template. As described earlier, the aspect construct was introduced so as to embody both links and multimedia objects that pertain to a particular "aspect" of a node. Such aspects can be added or removed at run-time, allowing for node properties to be acquired or lost dynamically. Upon visualization, a node instance will present itself with its associated aspects, whereas each aspect provides the necessary multimedia data to fill placeholders that are delegated to its aspect descriptor.

Aspects are utterly beneficial to data modeling, as properties described on an abstract aspect type level can be enforced across multiple, for the remainder unrelated, node types, independently of the node type inheritance hierarchy. Hence these properties can be packaged and inherited as a whole, which enables a dynamic and non-disjoint classification mechanism, relaxing the constraints of a rigid node typing hierarchy. At the *implementation* level, this also introduces a measure of modularization, such that different aspects to the same node can be coded independently. The main node cannot reach directly to the aspect's multimedia objects: it only offers a "forum" for the aspect to present its encapsulated content.

4.5.2 Link types/presentation routines as a node's interface

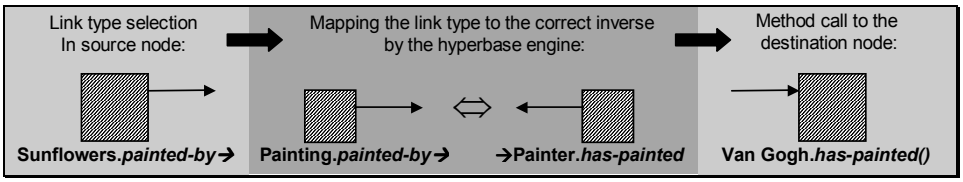
Whereas layout templates are designed without considering (relations to) other node types, the link types glue the different nodes together into a single network. This section denotes how a node's visualization is made *context-sensitive*, such that it reacts to a given link type by presenting the most relevant portion of its content.

Indeed, while the navigation paradigm deals with *inter-node navigation*, i.e. which node is made current at a given moment, most hypermedia environments also support the notion of *intra-node navigation*. The latter allows for a node not to visualize all of its embedded multimedia objects at the same time. By interacting with the node, the user is able to "navigate" between a node's internal multimedia objects (without moving to a different node altogether). A very simple example is scrolling within a single HTML document, but more sophisticated environments allow for multiple embedded multimedia objects to be visible at the same time, leaving the user the option of choosing between e.g. a picture of a painting or a textual description. For that purpose, we can introduce the term *internal node currency*. The latter denotes which of the current node's embedded multimedia objects is/are made current *within* this node. Internal node currency can be manipulated by the user by navigating

between the node's embedded objects. However, upon node access through an external link, the corresponding *presentation routine* is to determine how the node is visualized and which objects are to be made current initially. For that purpose, a node type can be equipped with as many presentation routines as it has link types. As a result, a node's link types determine its 'sensitivity' to different kinds of accesses, according to different reasons why it could be linked to the node from where it was accessed.

As described previously, a link's source node passes a link type ID as return value to the hyperbase engine upon closure. The hyperbase engine maps this link type unambiguously to an inverse link type, attributed to the destination node (see [18] for more details on the exact mapping mechanism). Thus, by acknowledging a node type's set of attributed link types as a factor in its visualization, it can provide an appropriate reaction to each situation in which it may be accessed. The presentation routine associated with the link type determines which subset of the node's multimedia objects, as assorted in its layout template, is to be made current upon node access through an instance of this link type. This allows a node to be sensitive to why it was accessed, such that the user can be directed to the most relevant section(s) of the node's information content. Of course, the concept of "being current", in the context of internal node information, depends on the implementation environment and multimedia data types involved. For visual objects, the presentation routine may be merely a matter of selecting a subsection of an HTML document. In richer environments, it may include scenarios for starting audio tracks, video sequences etc.

E.g. selecting the link type ***painted-by*** from the node **Sunflowers**, results in the node **Van Gogh** being accessed through a presentation routine associated with its own ***has-painted*** link type.



Again, this approach has the advantage that all required behavior is encapsulated within the node objects. Moreover, it allows for visualization properties to be laid down once on an abstract level: both layout template and attributed link types are node *type* or aspect *type* properties. Consequently, similar nodes will present a similar reaction to similar link type accesses. However, where necessary, general properties can be inherited and overridden to provide for a more specific reaction by means of *link subtyping*. A sub link type models a more specific relationship between two nodes than its parent, potentially provoking a more specific reaction by the destination node, by means of a more specific presentation routine associated with the link type.

Note that the complete layout of a node, i.e. what multimedia objects it should contain and how they should be presented, is designed in its layout template, independently of its link types. These only determine which part of this layout is to be made current upon node visualization. The node's layout definition is also to provide a user

interface, so as to enable the user to browse through the rest of the node's multimedia content, i.e. the data *not* made current upon access through the link type involved. Until the node is closed, all navigational control lies within the node's code, the hyperbase engine has nothing to do with this. Therefore, a node's user interface and "internal" navigation can be encapsulated, implemented and tested on their own, independently of the rest of the hyperbase.

5 Evaluation and conclusions

5.1 Nodes as self contained, independently coded entities

As explained above, because a node encapsulates all behavior necessary for its own visualization, it can be coded and tested independently, without prior knowledge of the other nodes it will interact with. The designer of a node (type) doesn't need to worry about which node types might be related to the one he is editing, the only criterion is the interface defined by attributed link types and their corresponding presentation routines. The destinations are irrelevant for node design. As a consequence, node design and maintenance can easily be delegated to different, independent parties.

Multimedia objects are hidden in the node's implementation and are never to be referenced directly from the outside world. Only a node's own presentation routines are allowed to access the multimedia information encapsulated within the node. Nevertheless, the *context-sensitive visualization* mechanism enables a node to present itself differently depending upon why it was accessed. Again, it only needs to have knowledge of and react to its attributed link types. If a link is selected to access a node, its inverse type becomes the criterion for the called node to determine its response. This allows the node to react not as much to *by whom* it is accessed, but rather to *the reason why* it is activated.

The layout template on node type level is able to describe both multimedia objects and properties for visualizing these objects when a node instance is accessed. At run-time, node visualization as determined in the node type's layout template is influenced by two additional elements. The first one is the *link type* through which the node was accessed. The second element is the node's associated *aspects*, allowing a single node to be attributed dynamically to multiple classes, whereas part of the visualization properties is determined in aspect layout templates. Aspects allow for specific multimedia information pertaining to a given node to be encapsulated in separate entities, possibly with their proper layout specifications. The appropriate aspects can be called and presented *at run-time*, again depending on the context in which the node is accessed. The combination of the context-sensitive visualization and self-containment principles results in each component (i.e. nodes as well as aspects) autonomously responding to a node access to the best of its ability.

5.2 Loosely coupled, heterogeneous nodes

The definition of nodes as self-contained entities and the separation of *node content* from the hyperbase's *link structure*, along with the introduction of a dedicated link storage facility, allows for *internal node maintenance* to be decoupled from *inter-node maintenance*. The former can, as discussed above, be executed independently of the other nodes, based on the node type's (and aspect type's) attributed link types. The latter consists of relational database queries on the *linkbase/repository* and does not affect node content. Moreover, the meta information stored along with the link data allows for automated completeness and consistency checking during authoring, or even the suggestion of feasible destination nodes for a given (source node, link type) combination. Obviously, easy accessibility of meta information at runtime is also a prerequisite for the *context-based navigation paradigm*.

Nodes are treated as real *objects* by the hyperbase engine, referred to by unique *object identifiers*, independent of their physical location. Where necessary, the hyperbase engine can generate a *navigation panel* at runtime with a complete node overview. The latter will also be useful for "third party" nodes that do not provide their own user interface for selecting navigational actions. Therefore, the specification of the node concept can remain very loose, and their actual implementations in a single hyperbase are allowed to be very diverse. The attributed link types play an interfacing role and allow for standardized interaction between the potentially heterogeneous node implementations.

A node does not embed direct references to other nodes. Rather, it passes an abstract link type ID representing the corresponding navigational action to the hyperbase engine upon node closure. The latter calculates the correct destination node by means of queries to the linkbase and retrieves its physical address. After that, the link type is mapped to its inverse and the destination node is accessed by means of the presentation routine that corresponds to this inverse link type. These link types are the only node properties that are to be known outside the node, they play the role of a node's *public interface*.

5.3 Specification on type level

A last advantage to the proposed approach is the fact that the majority of the properties can be defined on type level. Node layout as well as "source anchors" (i.e. the association between user interface events and link types) and "destination anchors" (i.e. the presentation routines) can be laid down in layout templates, associated with a node type. By means of inheritance and overriding, abstract specifications can be refined at more concrete levels of the typing hierarchy. In this respect, delegation to aspect types allows for specialization according to different criteria, such that a cohesive set of properties can be encapsulated into a single entity.

The fact that link "anchors" can be specified on type level again facilitates the separation of intra node maintenance from inter node maintenance. Link instances can

be updated without affecting their type level anchors. Only at runtime, such anchors are mapped to actual link instances by means of linkbase queries by the hyperbase engine. Needless to say that specification of properties on an abstract level will also improve consistency of layout and anchors, which in its turn reduces cognitive overhead and, consequently, end user disorientation.

5.4 Current prototype

MESH's web-based prototype is still in an experimental stadium. It consists of a *runtime environment*, based on a hyperbase servlet which processes navigational actions and accesses the linkbase/repository. The hyperbase engine is called from a "traditional" web browser. Nodes and aspects are conceived as static HTML or XML fragments, combined with a generic client-side applet which provides the necessary user interface functionality to handle the selection of navigational actions. These actions consist of both within-tour navigation and the initiation of new guided tours, as required by the navigation paradigm. The applet also transforms the appropriate HTML fragments into a single node visualization, according to the context in which it is accessed.

The HTML code only defines a node's encapsulated multimedia content: the links are removed from the node's content and are stored in the linkbase/repository, along with the meta information.

At present, the runtime environment provides a read only system: authoring is executed by means of a separate, offline, application. In the future, however, the runtime system is intended to enable user with the right privileges to reallocate links and update node content during a navigation session.

6 References

1. Ashman H., Garrido A. and Oinas-Kukkonen H., Hand-made and Computed Links, Precomputed and Dynamic Links, *Proceedings of Hypertext - Information Retrieval - Multimedia (HIM '97)*, Dortmund (Sep. 1997)
2. Bapat A., Wäsch J., Aberer K. and Haake J., An Extensible Object-Oriented Hypermedia Engine, *Proceedings of the seventh ACM Conference on Hypertext (Hypertext '96)*, Washington D.C. (Mar. 1996)
3. Bernstein M., The Navigation Problem Reconsidered, *Hypertext/Hypermedia Handbook*, E. Berk and J. Devlin Eds., *McGraw-Hill*, New York (1991)
4. Cockburn A. and Jones S., Which way now? Analyzing and easing inadequacies in WWW navigation, *International Journal of Human-Computer Studies No. 45* (1996)
5. Davis H., Hall W., Heath I., Hill G. and Wilkins R., MICROCOSM: An Open Hypermedia Environment for Information Integration, *Computer Science Technical Report CSTR 92-15* (1992)
6. Davis H., To Embed or Not to Embed, *Commun. ACM Vol. 38*, No. 8 (Aug. 1995)

7. De Bra P., Houben G. and Kornatzky Y., An Extensible Data Model for Hyperdocuments, *Proceedings of the fourth ACM European Conference on Hypermedia Technology (ECHT '92)*, Milan (Dec. 1992)
8. Furuta R. and Stotts P., The Trellis Hypertext Reference Model, *Proceedings of the Workshop on Hypertext Standardisation, Special Publication SP500-178,* National Institute of Standards and Technology, Gaithersburg (Jan. 1990)
9. Garzotto F., Paolini P. and Schwabe D., HDM - A Model-Based Approach to Hypertext Application Design, *ACM Trans. Inf. Syst. Vol. 11*, No. 1 (Jan. 1993)
10. Garzotto F., Mainetti L. and Paolini P., Hypermedia Design, Analysis, and Evaluation Issues, *Commun. ACM Vol. 38*, No. 8 (Aug. 1995)
11. Ginige A., Lowe D. and Robertson J., Hypermedia Authoring*, IEEE Multimedia Vol. 2*, No. 4 (Winter 1995)
12. Halasz F., Reflections on NoteCards: Seven Issues for Next Generation Hypermedia Systems, *Commun. ACM Vol. 31*, No. 7 (Jul. 1988)
13. Halasz F. and Schwartz M., The Dexter hypertext reference model, *Commun. ACM Vol. 37*, No. 2 (Feb. 1994)
14. Isakowitz T., Kamis A. and Koufaris M., The Extended RMM Methodology for Web Publishing, *Working Paper IS-98-18, Center for Research on Information Systems*, 1998 (Currently under review at ACM Trans. Inf. Syst.)
15. Jacobson I., Christerson M., Jonsson P. and Övergaard G., Object-Oriented Software Engineering, *Addison-Wesley*, New York (1992)
16. Knopik T. and Bapat A., The Role of Node and Link Types in Open Hypermedia Systems, *Proceedings of the sixth ACM European Conference on Hypermedia Technology (ECHT '94)*, Edinburgh (Sep. 1994)
17. Lange D., An Object-Oriented design method for hypermedia information systems, *Proceedings of the twenty-seventh Hawaii International Conference on System Sciences (HICSS-27)*, Hawaii (Jan. 1994)
18. Lemahieu W., Improved Navigation and Maintenance through an Object-Oriented Approach to Hypermedia Modeling, *Doctoral dissertation (unpublished)*, Leuven (Jul. 1999)
19 Lemahieu W., A Context-Based Navigation Paradigm for Accessing Web Data, *Proceedings of the ACM Symposium on Applied Computing (SAC 2000)*, Como, Italy (Mar. 2000)
20. Lemahieu W., *MESH*: A Model-Based Approach to Hypermedia Design, in: *Chen, Q. (ed.) Human Computer Interaction: Issues and Challenges*, Idea Group Publishing, Hershey, PA (Jan. 2001)
21. Meyrowitz N., Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework, *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '86)*, Portland (Sep. 1986)
22. Nanard J. and Nanard M., Hypertext Design Environments and the Hypertext Design Process, *Commun. ACM Vol. 38*, No. 8 (Aug. 1995)
23. Ramaiah C., An Overview of Hypertext and Hypermedia, *International Information, Communication & Education Vol. 11*, No. 1 (Jan. 1992)
24. Schwabe D. and Rossi G., Developing Hypermedia Applications using OOHDM, *Proceedings of the ninth ACM Conference on Hypertext (Hypertext '98)*, Pittsburgh (Jun. 1998)

25. Snoeck M., Dedene G., Verhelst M. and Depuydt A., Object-Oriented Enterprise modeling with MERODE, *Universitaire Pers Leuven*, Leuven (1999)
26. Thüring M., Haake J., and Hannemann J., What's ELIZA doing in the Chinese Room - Incoherent Hyperdocuments and how to Avoid them, *Proceedings of the third ACM Conference on Hypertext (Hypertext '91)*, San Antonio (Nov. 1991)
27. Thüring M., Hannemann J. and Haake J., Hypermedia and Cognition: Designing for comprehension, *Commun. ACM Vol. 38*, No. 8 (Aug. 1995)
28. Trigg R., Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment, *ACM Trans. Office Inf. Syst. Vol. 6*, No. 4 (Oct. 1988)
29. Wiil U. and Leggett J., Hyperform: a hypermedia system development environment, *ACM Trans. Inf. Syst. Vol. 15*, No. 1 (Jan. 1997)