

Encapsulation and Information Hiding as the keys to Enhanced Hypermedia Development and Maintenance

Wilfried Lemahieu

Department of Applied Economic Sciences

Katholieke Universiteit Leuven

Naamsestraat 69 B-3000 Leuven

Belgium

tel: + 32 16 32 68 70

fax: + 32 16 32 67 32

e-mail: wilfried.lemahieu@econ.kuleuven.ac.be

Keywords

Hypermedia, World Wide Web, object-orientation, navigation

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.

1 Introduction: object orientation and hypermedia

1.1 Hypermedia design based on a conceptual data model

Hypermedia systems represent data as a network of *nodes*, interconnected by *links*. The information embodied within the nodes can be accessed by means of *navigation* along the links, whereby a user's current position in the information space determines which information can be accessed in the next navigation step. This property of *navigational data access* raises hypermedia systems as utterly suitable to support user-driven exploration and learning. The user autonomously determines the way in which he will delve into the information, instead of being confined to the rigid "linear" structure of e.g. pages in a book.

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. [7], [14]: 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 [43]. 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 [17]. 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' *content*.

Whereas manually created links already reduce maintainability to a great extent, they also have a disastrous impact upon *consistency* and *completeness* [2]. 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 [26]. They yield richer domain knowledge specifications and more expressive querying. Typed nodes and links offer increased consistency in both node layout and link structure [31], [49]. Higher-order information units and perceivable equivalencies (both on a conceptual and a layout level) greatly improve orientation [25], [50]. Semantic constraints and consistency can be enforced [2], [24], tool-based development is facilitated and reuse is encouraged [42].

Consequently, hypermedia design is to be based on a firm *conceptual data model*. The pioneering conceptual hypermedia modeling approaches such as *HDM* [23] and *RMM* [28] 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* [16], *Hyperform* [53] and *Hyperstorm* [4]. Other approaches, such as *EORM* [33], *OOHDM* [44], [45] and *WebML* [13] modeled the *application domain* by means of the object-oriented paradigm.

1.2 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* [34]. In contrast to e.g. *OOHDM* [44], [45], *W2000* [5], *UHDM* [6] or *UWE* [32], *MESH* does not draw a sharp line between conceptual design and navigation design. Based on its conceptual modeling abstractions,

it offers a *context-based navigation paradigm* to accommodate for enhanced user orientation. Also, thanks to this navigation paradigm, the navigation model can be considered as a refined, evolved version of the conceptual domain model, whereas typical navigational constructs such as guided tours and indexes are generated at runtime. As a consequence, navigation really takes place in “conceptual space” which, as argued in [37], greatly facilitates orientation.

Therefore, *MESH* uses the same terminology at the level of conceptual design and navigation design: *node types* and *link types*, which will carry both a semantic and a navigational connotation. Its 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 [39], [47] 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. If desired, such a “logical” hypermedia model can be derived from a UML conceptual model, however, the semantic richness of *MESH*'s data model is adequate to directly capture conceptual domain specifications into a *MESH* model. Furthermore, the data model provides for a firm hyperbase structure and an abundance of meta-information that facilitates implementation of the context-based navigation paradigm.

Section 2 briefly discusses *MESH*'s data model, navigation paradigm and implementation architecture. A more elaborate description of the data model and navigation paradigm can be found in [35] and [36] 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*. The main focus of this paper is upon how encapsulation and information hiding further facilitate design and maintenance by separating a node's *interface* from its *implementation*. At the *conceptual* level, this allows for nodes to be considered as independent entities, which can be developed in parallel by different parties. Any node can be designed internally without the need for knowing the entire hypermedia structure. At the *implementation* level, it allows for nodes to be considered as heterogeneous components with a very loose coupling. They interact by means of well-defined public interfaces: their set of attributed link types, but can stay unaware of one another's actual implementation. As a consequence, each node or link can be updated without affecting the rest of the hyperbase, which obviously reduces the maintenance problem to a great extent. These issues are discussed in section 3. Section 4 draws comparisons to related work. Finally, conclusions and future research topics are presented in section 5.

2 An overview of the MESH framework

2.1 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*. The 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, this approach does certainly not reduce the granularity of node interrelations to referencing entire nodes because 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 (n_s, n_d) represents a one-to-one association between a *source node* n_s and a *destination node* n_d , 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). 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* (figure 1).

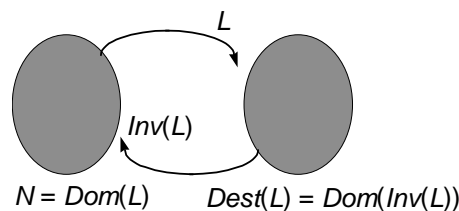


Figure 1: Link type domain, destination and inverse

At a conceptual level, node types and link types can be regarded as the equivalent of object types and association types in general-purpose object-oriented approaches. Note that a link *instance* always represents an association between exactly two nodes n_s and n_d . One-to-many and many-to-many associations are represented at *link type* level, depending on a link type's *maximum cardinality* (unique/non-unique): a *non-unique* link type L allows for multiple link instances of the same link type to share the same source node n_s : $(n_s, n_{d1}), (n_s, n_{d2}), (n_s, n_{d3}) \in L$. In this way, the combination of source node n_s and link type L yields a set of destination nodes $n_s.L := \{ n_{d1}, n_{d2}, n_{d3}, \dots \}$. The maximum cardinality of the inverse link type L^{-1} determines whether the association is one-to-many or many-to-many. On the other hand, if L is a *unique* link type, $n_s.L$ will be a singleton. The maximum cardinality of the inverse link type L^{-1} determines whether the association is one-to-one or many-to-one. Link type properties such as *domain*, *minimum cardinality* (optional/mandatory), *maximum cardinality* (unique/non-unique) and *destination/inverse* allow for enforcing constraints on their instances. These properties can be overridden to provide for stronger restrictions upon inheritance. For example 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*.

2.2 The use of aspects and aspect descriptors to overcome limitations of a rigid node typing structure

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 [35] for a more thorough discussion). However, to adequately describe the information in a hypermedia system, a single classification criterion is very limiting. 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 able to change over time. Also, aspects can provide an elegant solution in many situations that would otherwise call for multiple inheritance.

Aspects are defined as instances of *aspect types*. To a certain extent, aspect types can be compared to *role types* in the object-oriented paradigm, e.g. [52], in that these are associated with a “main” object type, to which they model additional, specific behavior. In the same way, aspect types are inextricably associated with a “main” node type. However, their purpose is different. Role types are primarily meant to enforce *sequence constraints* upon multiple, concurring life cycles of the same object type. Such semantics are not present for aspects. The goal of the aspect construct is to allow for an additional, non-disjoint and dynamic node classification mechanism, resulting in “volatile” node properties. In this way, aspect types are also comparable to subtypes, in that they are able to *override* properties that are defined in the main node type.

The actual supplementary classification is accomplished through *aspect descriptors*. 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/frozen*. Depending on these properties, a node instance can/must have one/many values for the aspect descriptor, which may/may not change over time. Each possible value of an aspect descriptor is associated with an aspect type. This 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, according to the classification criterion that is represented by the aspect descriptor.

A node instance may carry multiple aspects and can be described by as many aspect descriptors as there are additional classification criteria for its node type. As opposed to node types, aspects are allowed to be volatile: dynamic classification can be accomplished by manipulating aspect descriptor values, thus adding or removing aspects to a node at run-time.

Aspect types feature the same properties as node types: *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. Aspect types are able to override (i.e. refine) their “main” node type’s layout and link type specifications. The inheritance/overriding mechanism is similar to the mechanism for “real” node supertypes/subtypes, but because an aspect descriptor can be multi-valued, particular care was taken so as to preclude any inconsistencies that could result from non-disjoint classification (see [35] for further details). For instance to model an **artist** that can be skilled in multiple disciplines, a multi-valued aspect descriptor *discipline* defines a non-disjoint classification over **artist**, resulting in the **painter** and **sculptor** aspect types. Discipline-specific node properties are modeled in these aspect types. An **artist** node that has {painter, sculptor} as the values to its *discipline* aspect descriptor, e.g. the **Michelangelo** node, features the combined properties of its **Michelangelo.asPainter** and **Michelangelo.asSculptor** aspects (figure 2).

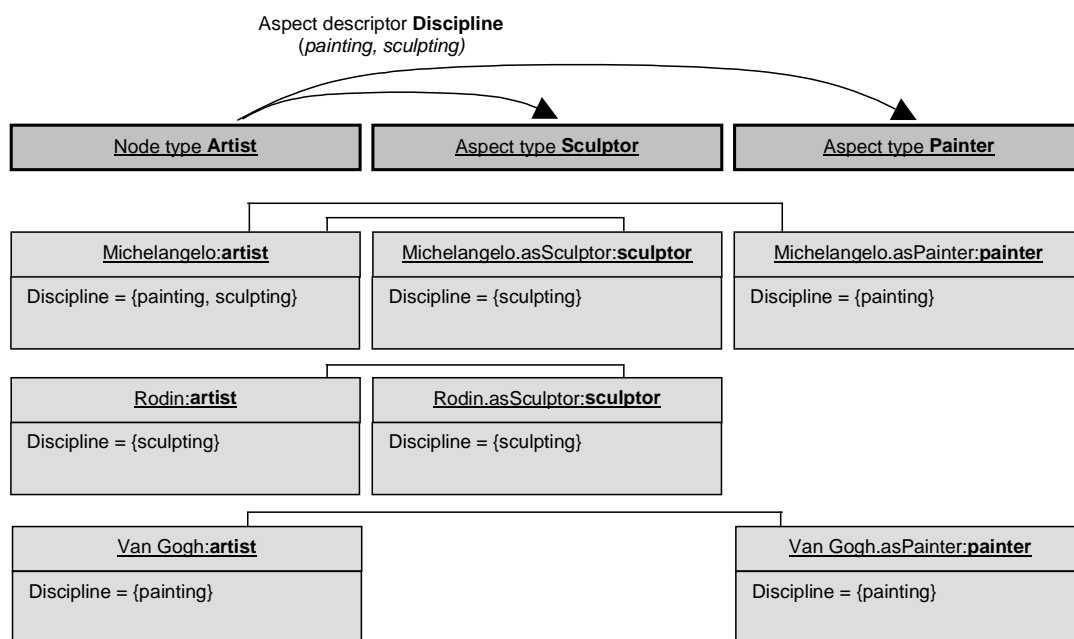


Figure 2: Aspect descriptor and aspect type

If the *discipline* aspect descriptor allows for a node to change its aspect descriptor value(s) over time, i.e. it is not frozen, an **artist** node will be able to “gain” (or lose) additional disciplines at runtime and

the classification (and node properties) becomes dynamic. Therefore, at the implementation level, aspects can be seen as *components* that are plugged into the “main” node object. In this way, aspects can be used as real building blocks for nodes. Link types and layout definitions pertaining to a single “role” a node may have to play are encapsulated into one aspect type, which allows for the modeling of a similar ‘aspect’ in otherwise completely dissimilar node types. In this way, different aspects associated with the same node instance can even have different editing privileges, such that updating multimedia *content* can be delegated to different parties, each responsible for a particular aspect of the node.

2.3 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. Therefore, *MESH* considers link types as full-fledged abstract types that can equally be subject to subtyping.

Moreover, link types play an utterly important role in each stage of the development of *MESH* based hypermedia applications. Their properties such as domain, cardinalities and inverse allow enforcing semantic constraints over node instances. At a conceptual level, they represent *semantic relationships* between the node types, which represent *domain classes*. At a navigational level, instances of the link type define the *navigation paths*, hence they make out the actual hypertext structure. At the implementation level, they are stored as tuples in the linkbase (cf. infra) and they determine the transitions between the different software components that make out the nodes’ implementations. Moreover, the link types are used as input parameters to the nodes’ visualization routines, to cater for *context-sensitive node visualization*, as discussed further on in this paper. 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.

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 . Moreover, L_c 's properties (i.e. domain, minimum cardinality, maximum cardinality and destination/inverse) can be overridden to provide for more specific semantic constraints than the ones enforced by L_p . If L_c overrides L_p 's domain, the specialization is called *vertical*: it is the consequence of a parallel classification over the link types' domains (through subtyping or through an aspect descriptor), 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 does not override L_p 's domain, i.e. they 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.

Examples of both horizontal and vertical link subtyping are presented in figure 3: if the **painter** node type is a subtype of **artist**, the *has-made* link type leading to the artist's creations can be inherited and overridden in **painter**, to become the *has-painted* link type, which obviously entails a more specific semantic meaning than the parent link type. Because the domain of *has-painted* is a subtype of *has-made*'s domain, the specialization is *vertical*. However, if the node type **artist** features a link type *exhibited-in*, leading to **museum** nodes that exhibit some of the artist's work, one could imagine a child link type *dedicated-exhibitions*, leading to only these museums that have an entire exhibition dedicated to that particular artist, which obviously again entails a more specific semantic meaning than

the parent link type. However, this link type specialization is independent of any node type specialization over **artist**. Both parent and child link type have the same domain, hence the link specialization is *horizontal*. Note that, for a horizontal as well as a vertical link type specialization, instances of the child link type comply with the parent link type's constraints, but may carry more specific semantics than the parent's instances, possibly reflected in stronger constraints for the child link type.

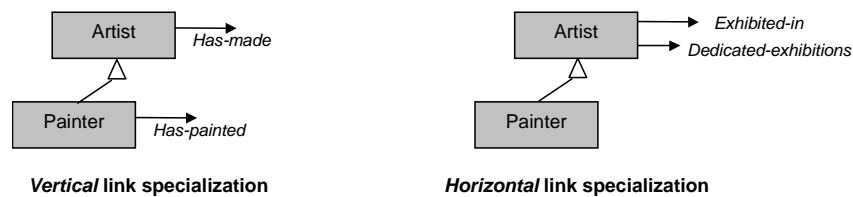


Figure 3: Horizontal and vertical link specialization

Apart from the domain, a link type's cardinalities and inverse can be overridden as well upon specialization. *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 [35].

2.4 MESH's context-based navigation paradigm

2.4.1 Guided tours derived from the current context

MESH's navigation paradigm is discussed thoroughly in [36], this section offers a brief overview. The navigation paradigm combines set-based navigation principles [1], [19], [30] 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 [51].

In conventional hypermedia applications, the *current node* is the only variable that determines which information is accessible at a given moment; navigation is only possible to nodes that are linked to this current node. Its value changes with each navigation step as it represents the immediate focus of the

user's attention. *MESH* introduces the *current context* as a second, longer-term variable that 'glues' the various visited nodes together and provides a background about which common theme is being explored. The current context is defined as the combination of a *context node* and a *context link type*. The context node represents the subject around which the user's broader information requirements 'circle'. The nature of the relationship involved is depicted by the context link type.

A guided tour derives from the current context. Therefore, *MESH* discriminates between *direct* and *indirect* links. A direct link represents a lasting relation between two nodes. Direct links are typed and reflect the underlying conceptual data model. Because they are permanent and context-independent, they are stored explicitly into the hyperbase and are always valid. E.g. the node **Sunflowers** is directly linked to the **Van Gogh** node. An *indirect* link between two nodes indicates that they share relevancy to a common third node. The latter denotes the *context* within which the indirect link is valid. As indirect links not only reflect the data model, but also depend on a run-time variable, the *current context*, they cannot be stored within the hyperbase. They are to be created *dynamically* at run-time, as inferred from a particular context. E.g. an indirect link between **Sunflowers** and **Wheatfield** is relevant when exploring information related to Van Gogh but not when retrieving information with regard to "paintings of flowers".

As opposed to traditional guided tour implementations, which are hard-coded into the hyperbase, *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. A guided tour is defined as a path of *indirect* links along all nodes relevant to the current context. These nodes are directly linked to the context node (through instances of the context link type) and indirectly to their predecessor and successor in the tour. As they are chained into a linear structure, a logical order should be devised in which the subsequent tour nodes can be presented to the user. The most obvious criterion is in alphabetical order of a *node descriptor* field. More powerful alternatives, e.g. where the user can provide ordering criteria or where the system can suggest an "optimal" ordering based on previous navigation steps, are discussed in [34]. For instance in figure 4 the context **Van Gogh.has-painted** yields a guided tour (represented as a

dotted line) among the nodes { **Irises**, **Potato eaters**, **Starry night**, **Sunflowers**, **Wheatfield**, ... } with **Van Gogh** as the *context node* and *has-painted* as the *context link type*.

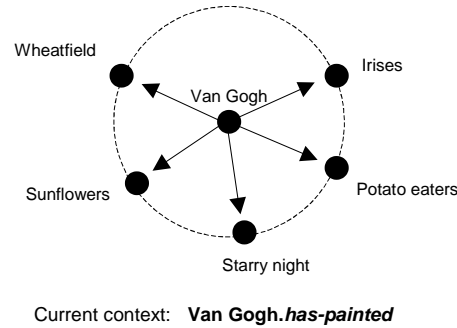


Figure 4: Example of context and guided tour

2.4.2 Navigation within and orthogonal to the current tour

Navigation in *MESH* can be classified according to two dimensions. First, there is moving *forward* and *backward* within the current tour, along indirect links. Second, and orthogonal to this, there is the option of moving *up* or *down* along direct links, closer to or further away from the session’s starting point. The former provides “linear” assistance to a disoriented user, the latter offers the navigational freedom that is the trademark of hypertext systems.

Moving forward or backward in a guided tour along indirect links, results in the node following/preceding the current node being accessed to become the new current node. The current context is unaffected (figure 5).

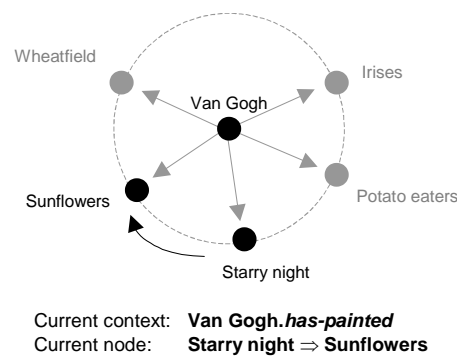


Figure 5: Navigation within the current tour

Moving down implies an action of ‘digging deeper’ into the subject matter, looking for additional information regarding the current node. This is accomplished through selection from the current node of a direct link type. In the case of a *unique* destination node, the result is the latter node being accessed. In the case of a *set* of destination nodes, the outcome is a new nested tour being started.

Indeed, instead of selecting a single *link instance*, similarly to the practice in conventional hypermedia, a navigational action consists of selecting *an entire link type*. Selection of a *unique link type* results in a single destination node being accessed, e.g. the link type selection **Sunflowers.exhibited-in** results in the node **National Gallery** becoming the new current node. The context is unaffected.

Selection of a *non-unique link type* from a given source node effects into a *guided tour along a set of nodes* being generated. Such action induces a *context change*: a new context emanates, resulting in new indirect links. The result is a new guided tour, nested within the former, generated in accordance with the new context. The tour includes all nodes that are linked to the given node by the selected link type, e.g. selection of the link type **Sunflowers.reviews** results in the guided tour {**review #1, review #2, review #3...**}. The current node **Sunflowers** is denoted as the new context node. The non-unique link type *reviews* defines the context link type. The first review is accessed to become the new current node. Such *context change* reflects the user’s decision to concentrate on the current node as a new topic of interest. All indirect links are destroyed and redefined around this new context (figure 6).

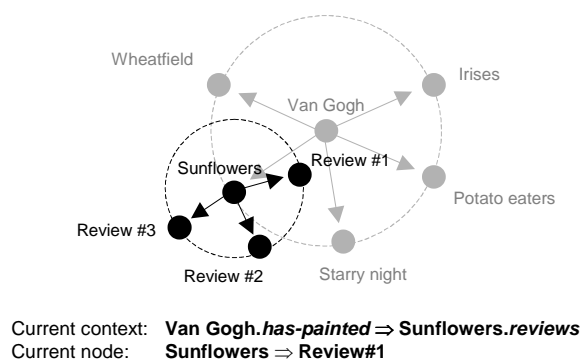


Figure 6: Navigation orthogonal to the current tour

Hence contexts, and consequently guided tours, can exist in layers. As such, it is possible to ‘delve’ into a subject and have multiple *open* tours, nested within one another, where the context node of one

tour is the current node of the tour it is nested in. Navigation along indirect links is invariably carried out within the “deepest”, i.e. most recently started tour. Continuing a tour on a higher level is only possible if all tours on a lower level have been either completed or disbanded. The latter is accomplished by *moving up*, which reverses the latest *move down* action. If the latter involved a context change, the move up action results in the reestablishment of the previous context and the cancellation of the tour generated through this most recent link type selection. The previous context’s context node and indirect links are restored. In this way, the user perceives the network structure in a very natural way as a *hierarchy*, which as discussed in [9] greatly facilitates comprehension. However, in *MESH*, such hierarchic view is neither static nor pre-defined, but develops gradually as the user explores the information.

2.4.3 Abstract navigational actions defined as link type selections

Both forward/backward and up/down types of navigation can be described as *abstract navigational actions*, i.e. they can be described as link *type* selections but are independent of the actual link *instance(s)* involved. The latter ones can be calculated automatically at runtime, based on the current context and the direct links stored in the hyperbase. Moreover, the practice of node and link typing allows for casting navigational actions to a whole *class* of nodes, regardless of the actual instance they are applied to. In this way, selections of link *types* that exist at a sufficiently high level of abstraction can be imposed upon every single node belonging to a tour. E.g. in the context of **Van Gogh.has-painted**, a **painting#x.reviews** selection can be issued once on *tour* level, with additional (nested) tours being generated automatically for each node participating in the **Van Gogh.has-painted** tour. If these tours in their turn include navigational actions on type level, a complex navigation pattern results, which can be several levels deep. Still, it is perceived by the end user as one giant guided tour, which actually consist of multiple, nested subtours. The entirety of these tours can be traversed sequentially.

The abstract navigational actions and tour definitions sustain the generation of very compact overviews and maps (either hierarchic or network-based) of complete navigation sessions. Such maps can be based solely on high-level *navigational actions* and currency indications of open tours, without every single *node* participating in the various tours being retained. The same information can also be used in

bookmarks, i.e. bookmarks in *MESH* do not just refer to a single node but to a complete *navigational situation*, which can be stored and resumed at a later date. Most important to this paper, however, is that each navigational action can be described in terms of a *link type*: navigation *within* the current tour is defined by the context link type. Navigation *orthogonal* to the current tour can be described by the newly selected (unique or non-unique) link type.

2.5 A platform-independent implementation framework

2.5.1 Separation of navigation structure from node layout

Although in theory *MESH*'s data model and navigation paradigm can be implemented above any physical hypermedia architecture, this section presents one possible implementation framework that fully supports all of *MESH*'s facets.

Indeed, 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, as 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 such as OQL [12].

However, such “universal storage” approach, 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 “universal access” option leaves much more freedom to implement the *content* of a node.

2.5.2 Stages in MESH based hypermedia design

As discussed further in section 5.3, *MESH*'s current runtime application consists of a read-only environment. Authoring is to be accomplished through a separate, offline design environment. Nevertheless, the above mentioned relational database plays the role of a *linkbase/repository*, which furnishes the necessary metadata and linkage information during the design phase as well as at runtime. This subsection discusses the different stages in *MESH* based hypermedia design, whereas the runtime application architecture is discussed in the subsequent subsection. Both issues, as well as the *linkbase/repository*'s vital role in them, are summarized in figure 7.

As mentioned previously, *conceptual/logical design* can be based directly on *MESH* constructs or can be accomplished by means of a preliminary UML class diagram, which is then translated into a *MESH* model. The resulting specifications of node types, link types, aspect descriptors and aspect types are stored in the *linkbase/repository*. Dotted lines in figure 7 represent information being transferred to the *linkbase/repository* at design time.

Conceptual/logical design:

(Direct input from requirements analysis or from an existing UML model)

Specification of metadata, to be stored into the linkbase/repository:

- Node types (cf. conceptual classes)
- Link types (cf. association types)
- Aspect descriptors (cf. additional classification criteria)
- Aspect types (cf. "subordinate" object types)

Navigational design:

- Refinement of the conceptual design, considering the instances of the node types will be the actual units of currency during navigation

Link implementation:

- Definition of link instances, to be stored in the linkbase/repository
- Navigation patterns can be tested, based on a system-generated navigation panel and information from the linkbase/repository, even before the nodes' actual content is implemented

Layout design:

(Define layout templates for each node type and aspect type)

- Determine placeholders for multimedia objects
- Determine how an aspect template's placeholders map to the "main" node's template
- Associate user interface events with navigational actions
- Associate a visualization routine with each link type

Node implementation:

(individual nodes are implemented as system components with a limited external interface)

- Create the nodes' or aspects' multimedia content in correspondence with their type's template
- Implement each node's visualization routines, based on its attributed link types
- Aspects are implemented as separate components that can be "plugged" into the main node
- This can be done for each node separately, without knowledge of other node or link instances

Runtime:

- Interaction between nodes, hyperbase engine and linkbase/repository

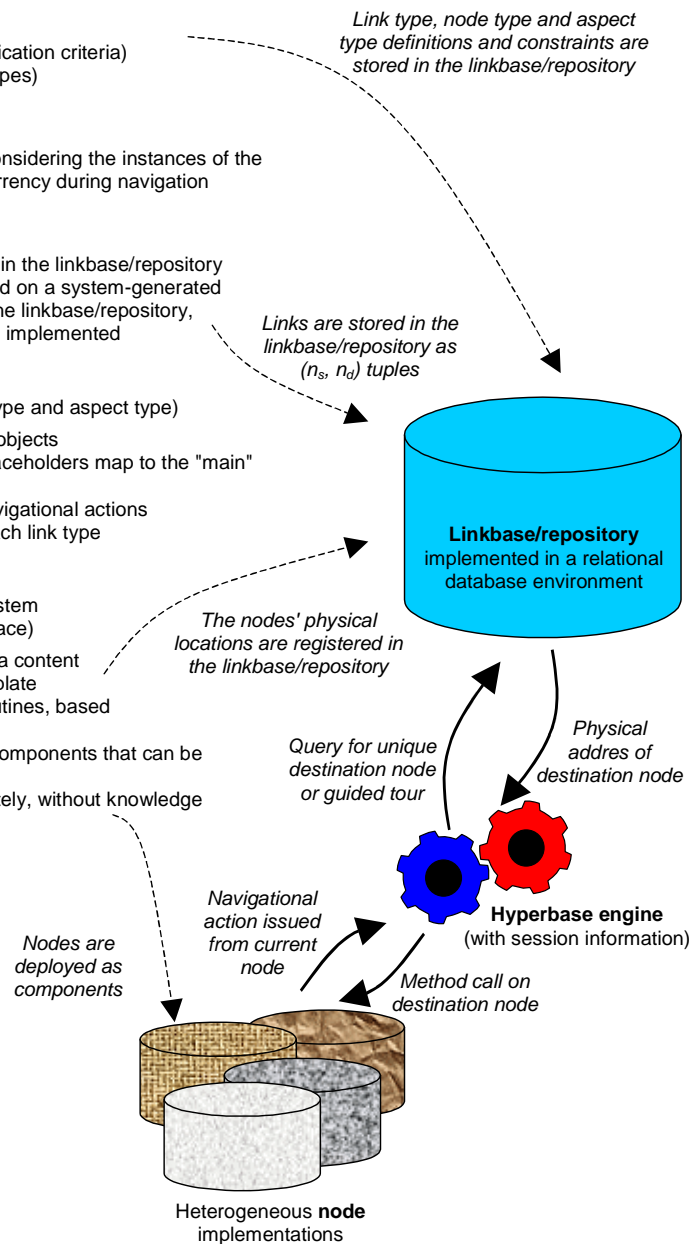


Figure 7: Development stages and runtime interaction for a MESH application

Because typical navigational constructs are generated automatically at runtime, the *navigational design* efforts can be kept to a strict minimum: the navigational design can be seen as a refinement of the conceptual design. For instance, conceptual classes whose instances should not be treated as real units of currency during navigation, can be subsumed into other conceptual classes to become a single node type. Once the navigational design is completed, the subsequent design efforts can be executed

concurrently and independently: on the one hand, *link instances* can be created in the linkbase/repository as (*source node*, *destination node*) tuples. Hence, these are stored separately from the nodes' actual *content*. This approach allows for the entire navigation structure to be developed and tested before the nodes themselves have been implemented. On the other hand, the design of a *layout template* for a given node type or aspect type can be carried out without knowledge of other node or aspect types: the only "exterior" information needed are the link types attributed to the node type or aspect type for which the template is being developed (see section 3.3.3 for further details). Afterwards, node and aspect instances can be coded in correspondence with their type's template. Nodes and aspects are deployed as loosely coupled *components* into the final system. Again, each individual node's behavior can be implemented and tested without knowledge of other nodes or the actual navigation structure, solely based on the link types' interfacing role.

2.5.3 MESH's runtime architecture

In broad outlines, *MESH's* implementation architecture and interaction mechanism correspond to the OHP framework proposed by the Open Hypermedia Community [18], [41]. The resulting system consists of three types of components: the *nodes*, the *linkbase/repository* and the *hyperbase engine*, all depicted in figure 7. The solid lines represent the actual runtime interaction mechanism. In [34], 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 from 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 *public interface*.

Since a node is not specified as a necessarily searchable object, linkage information cannot be embedded in a node's body. Links, as well as meta data about node types, link types, aspect descriptors and aspect types are captured within the 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 *database queries*, i.e. without the need for searching through *node content*. In its most rudimentary form, a query to calculate all nodes in a guided tour as resulting from selecting the link type *theSelectedLinkType* from the node *theCurrentNode* can be generated automatically and looks as follows (an actual query will be more complex as also child link types are to be taken into consideration):

```
select n.nodeID, n.nodeName, n.physicalLocation, lt.linkTypeName
from nodes n, links l, linkTypes lt
where l.linkType = :theSelectedLinkType_ID
and l.sourceNode = :theCurrentNode_ID
and n.nodeID = l.destinationNode
and lt.linkTypeID = l.linkType
order by n.NodeName [or some more complex ordering criterion]
```

The combination of the hyperbase engine and linkbase/repository closely matches the concept of a *linkservice* as defined in OHP. However, its task is more elaborate, as a crucial part of its functionality lies in keeping track of the current context and deducing guided tours from run-time information (the current context) and persistent information (stored links and metadata). Its is also to provide the

destination node with information about the link type through which it is accessed, which is essential to context-sensitive node visualization, as discussed further on in this paper.

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 public 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.

3 An object-oriented approach to node interaction

3.1 The anchor notion

The traditional concept of an *anchor*, as defined e.g. in [27], 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, e.g. to refer to a specific fragment in a text document or a specific sequence in a video file. 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 [27], [40], 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* [27]. Both have the same purpose: referring to internal components of a node's content. However, on a behavioral level, there certainly is a difference [22], [33]. 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. At this stage it is also possible, if required, to test for *preconditions* that determine whether the navigational action is allowed to take place at all. 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*. For instance 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. Hence the destination anchor determines the *postconditions* of the navigational action, i.e. the state in which the hypermedia system is left after completion of the action.

3.2 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 as it is traditionally defined, 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.

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

3.3.1 General principle

However, *MESH* reconciles the need for anchoring with the encapsulation concept by providing a truly object-oriented alternative: instead of externally exposing anchors to a node's internal multimedia objects, *MESH* uses a node's *attributed link types* to interface between the global hyperbase objects and the node's internal components. By calling upon the applicable interface method, the node autonomously determines the suitable multimedia content to be presented. The association of a user interface event with a navigational action can be seen as the equivalent of a *source anchor*: if a user interface object is suitably 'stimulated' by the user, the corresponding user interface event causes a *navigation step*. Because of *MESH*'s navigation mechanism, such navigational action will actually correspond to the selection of a *link type*. This link type 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.

The link type not only has an influence on *which node* will be accessed next, but also on *which visualization method* will be called upon this destination node. Indeed, 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. The two subsequent subsections further elaborate on *MESH*'s alternatives to source anchors and destination anchors respectively.

3.3.2 Link type selections instead of source anchors

It has already been 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 layout template. In this way, it provides the user with a means to interact with the node and explore its embedded multimedia objects. 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, again according to the encapsulation principle, left to its internal design. As 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 selection*. As the consequence of a navigational action being issued by the user, 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: the ID of the selected link type. 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 user interface 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). Hence maintenance of the individual link instances (in the linkbase) does not affect the node's internal properties and whenever a new node 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* and, eventually, destination nodes. In this way, selection of the link type L from the source node n_s results in a set of destination nodes $n_s.L := \{n_d \mid (n_s, n_d) \in L\}$ which is calculated at runtime.

A unique link type is mapped to a singleton, i.e. a unique destination node, e.g. **Sunflowers**.*Painted-by* := {Van Gogh}. A non-unique link type is mapped to a *guided tour*, of which the first participating node is accessed. Such a 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, e.g. **Van Gogh**.*has-painted* := {**Potato eaters**, **Self portrait**, **Sunflowers**,...}. These nodes can be visited sequentially by the user. Moreover, 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. Note that any **painter** node “anchors” the same action *has-painted*, independently of the actual node and link instances. It's the hyperbase engine that translates this action to a guided tour, based on the current context and the information in the linkbase/repository. This approach does not only facilitate development to a great extent, but also improves the user's grasp on 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 internal *node content*. In addition, such navigation panel can provide a user interface

to select navigational actions 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 [2], 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. All links representing relevant semantic associations between nodes are stored in the linkbase, even if they are not used as explicit navigation paths: they serve as input to generate appropriate guided tours. This is also partially a consequence of inverse links being automatically generated for each link added. For instance 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. Therefore, one of the few decisions to be made during navigational design is which of the links or link types attributed to a given node type (or aspect type) will actually be anchored. It’s the designer’s responsibility to weigh off the supply of relevant additional information against the risk of cognitive overload due to an overdose of link anchors. In the case of link types having subtypes, there is also the choice between anchoring only the parent link type, only the children or both parent and children. However, these decisions never affect navigational freedom, as all links are stored in the linkbase/repository. Even non-anchored links (which incidentally may represent associations relevant to the user) can be made visible by a system generated node overview. The only restriction is the possibility of certain deliberate *preconditions* preventing a given navigational action, as applied e.g. in computer based learning systems or simply because not all users have the same access rights.

3.3.3 Context sensitive visualization instead of destination anchors

3.3.3.1 A node type’s layout template

Because an external 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 subsection deals with the more general aspects of layout templates and node visualization. The subsection hereafter elaborates on the context-sensitive node visualization mechanism.

Indeed, as discussed in detail in [34], 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. This framework is unique for a given node type and is independent of the link types through which node instances will be accessed. 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. For example a template could be defined as an XML schema, 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. Synchronization among the multimedia objects can be accomplished e.g. by defining them as a SMIL presentation [3].

Just like link types, layout templates can be inherited and overridden in both child node types and aspect types. Intentionally, the description of the layout inheritance and overriding mechanism is kept very general and abstract in [34]. 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 link types 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. For that purpose, the "main" node's layout template is to define placeholders, some of which have their content delegated to a particular aspect descriptor. Aspect types that correspond to this aspect descriptor are to feature a layout template that determines how a delegated placeholder is to be filled. Instances of such aspect type contain the actual multimedia objects, which are visualized in the placeholders, along with the main node's own multimedia objects. If a node contains multiple values for the same aspect descriptor, multiple aspects will concur to fill the placeholders delegated to this aspect descriptor. For such multi-valued aspect descriptors, the main node's user interface is to provide a means to switch between different aspects of the same node. E.g. the **Michelangelo** node is to provide a mechanism for switching between its **Michelangelo.asPainter** and **Michelangelo.asSculptor** "appearance". Again, this can be defined e.g. by means of a SMIL *exclusive time container*.

Aspects are utterly beneficial to data modeling, as properties described on an abstract aspect type level can be packaged and inherited as a whole across multiple, for the remainder completely dissimilar, node types. At the *implementation* level, this also introduces a measure of modularization, such that different aspects to the same node can be coded as independent components that are "plugged" into the main node. The main node cannot reach directly to the aspects' multimedia objects: it only offers a "forum" for an aspect to present its encapsulated content.

3.3.3.2 *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 subsection 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. The latter can be seen as a generalization of the destination anchor

concept, with the added advantage of the actual implementation of this anchor being hidden from the outside world.

Indeed, many types of multimedia objects can be visualized in multiple ways. For instance a simple HTML page can be “scrolled” to a certain position when presented on screen or a video file can be started from a given fragment. In traditional environments, this is achieved by associating a destination anchor with the desired fragment. In more sophisticated environments, where multiple objects can be visualized on screen at the same time, a destination anchor may determine which (subset of) a node’s multimedia objects is displayed on screen in a given situation. In *MESH*, such destination anchors are not exposed to the exterior, but are only accessible from a node’s own internal *presentation routines*. Only the interface of these routines is accessible from outside, their implementation (by means of traditional anchors or otherwise) is hidden. However, upon node access through a link of a given type, i.e. within a given *context*, a suitable presentation routine is selected to visualize the node in a way that is appropriate in that particular context. A possible way of implementing this is the link type being used as input parameter to a SMIL presentation, hence influencing which multimedia objects are displayed and when.

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 [34] for more details on the exact mapping mechanism). Therefore, by providing a node type with as many presentation routines as it has link types, it can present an appropriate reaction to each context in which it may be accessed. The presentation routine associated with a link type determines which subset of the node’s (and corresponding aspects’) multimedia objects, as assorted in its layout template, is to be visualized 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. Hence, in contrast to e.g. *Hyperbase* [46], *MESH* is in no way limited to interlinking nodes in their entirety. Obviously, the actual implementation of this approach depends on the implementation environment and multimedia data types involved. The presentation routine may be merely a matter of selecting a subsection of an

HTML document or, in richer environments, it may include scenarios for starting audio tracks, video sequences etc.

For example, 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 (figure 8). This presentation routine may visualize a different subset of the Van Gogh node's content than e.g. a visualization routine associated with a **Van Gogh**.*native-village* link type.

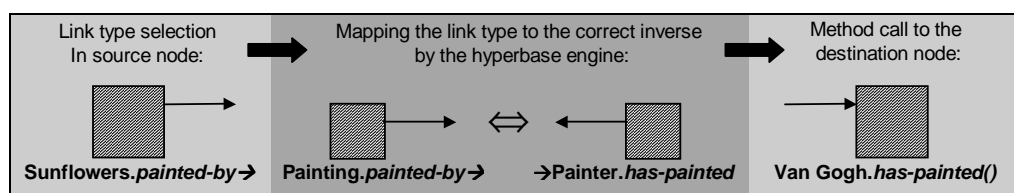


Figure 8: Mapping of a link type selection to a presentation routine

Again, this approach has the advantage that all required behavior is encapsulated within the respective nodes. 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.

To summarize, node interaction is regarded as interaction between *self contained objects*. Each node 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 link ID as *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. This clear separation of concerns between source node,

hyperbase engine and destination node results in a very loosely coupled system; the source node needn't be concerned with the destination of a link and vice versa, such that updates to one node (or to the link structure) by no means affect other nodes' content. This obviously greatly benefits maintainability, as well as the ability to delegate the initial development of individual nodes to independent parties, which only need to have knowledge of a common link type structure. Moreover, note that the current node is always fully responsible for its interaction with the user; minimally, it is to provide a means for specifying the next navigation step. 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 mechanism can be encapsulated, implemented and tested on their own, independently of the rest of the hyperbase.

4 Related Work

4.1 System architecture

As already explained at the beginning of the paper, object-orientation was initially applied in hypermedia to model *system components*, rather than the *conceptual domain*. Systems such as *Microcosm* [16], *Hyperform* [53] and *Hyperstorm* [4] take advantage of an object-oriented approach and offer a low-level data model that provides HBMS (hyperbase management system) functionality. The latter includes unique OID's, enforcing integrity constraints, offering concurrency control, crash recovery, versioning, support for collaboration, locking, access control, query functionality, multi user access, transaction management etc. They are often positioned between a presentation layer and an underlying general-purpose OODBMS or sometimes RDBMS. However, as these systems do not implement a specific data model, the object-oriented approach mainly pertains to the *components* of a hypermedia application, rather than offering an object-oriented view of the *domain model* that is to be represented in the application. Also, navigation semantics are not specified and are left to the designer of each specific application implemented within the HBMS.

Whereas earlier efforts envisaged a rather monolithic approach, these more recent systems set out from the premise that node content (i.e. multimedia objects) and links are two completely different concepts

in a hypermedia environment, with distinct manipulation and storage requirements. *MESH* has in common with such systems that the coupling between the different components is kept as loose as possible. This results in links being isolated from node content and being stored in a separate link database. Hence, *MESH* roughly shares the same principles on system level, as appointed by the Open Hypermedia System community in OHP [18], [41], but apart from that explicitly specifies semantics for conceptual data modeling and navigation. The latter forms an additional motivation for a separate link database, to be able to generate guided tours by means of relational database queries. Moreover, *MESH* explicitly stores all conceptual metadata, i.e. definitions of node types, link types, aspect descriptors and aspect types and all corresponding constraints, in its linkbase/repository. Its object-oriented approach is not restricted to the system components, but also applies to the hypermedia's *domain model*. The objects represented in the domain model will exist as components in the final system, along with the hyperbase engine. The loose coupling is, true to the object-oriented paradigm, accomplished by means of interfaces that hide encapsulated functionality. These interfaces correspond directly to link types in the conceptual model.

This explains another difference between *MESH*'s linkbase/repository and OHP's linkserver notion: neither anchors nor presentation specifications are stored within the linkbase. The way in which a source node generates navigational actions as link type selections and the way in which a destination node visualizes itself according to the link type through which it is accessed is considered an internal node property, to be encapsulated within a node's content.

4.2 Conceptual domain modeling and navigation

Whereas the object-oriented techniques used in the HBMSs described above are mainly aimed at modeling functional behavior of the system's components themselves and for supporting specific media types, i.e. to facilitate *implementation*, approaches such as *EORM* [33], *RMM* [28], *HDM* [23] and *OOHDM* [44], [45] have in common with *MESH* that the modeling of the *conceptual domain* is accomplished through object-oriented or entity-relationship techniques. Differently to said HBMSs (and also to most non hypermedia-specific object-oriented methodologies) *structure* and *relationships* prevail over *behavior* as important modeling factors. However, *MESH* is the only approach that

provides modeling primitives such as link subtyping and a node classification mechanism supporting plural, dynamic specializations by means of aspect descriptors and aspects. As a consequence, each node corresponds to a real world concept hence navigation in *MESH* closely matches navigating conceptual space as defined in [37]. The latter distinguishes between *hyperspace* and *conceptual space*. Hyperspace refers to the hypermedia structure itself, whereas conceptual space involves the actual concepts and interrelations represented in the hypermedia system. A *close correlation between hyperspace and conceptual space* is claimed to significantly advance comprehension and orientation.

EORM, *RMM*, *HDM* and *OOHDM* restrict themselves to a “standard” entity-relationship or object-oriented approach. *WebML* [13] provides a recent, model-driven approach to web site development. Complex web sites are initially specified at the conceptual level. Of all systems mentioned, *WebML*’s data model is the one that resembles *MESH*’s the most. At least on the level of individual node types, as *WebML* does not provide an inheritance mechanism, link subtyping nor secondary node classifications by means of aspect descriptors and aspects. Moreover, *WebML* does not provide its own data model but borrows from entity-relationship modeling and UML. Also worth noting in the context of UML are *W2000* [5] and *UHDM* [6]. These are positioned as frameworks for designing web applications, i.e. apart from hypermedia functionality they also deal with operations and transactions that affect node content. For that purpose, they extend UML by means of specific stereotypes and customizations of diagrams to accommodate for structural and navigational hypermedia abstractions. The underlying hypermedia constructs of *W2000* and *UHDM* are borrowed from *HDM* and *OOHDM* respectively. *UWE* [32] stands for a UML-based approach to Web application design as well, with particular emphasis on the authoring process. Combining hypermedia constructs from *RMM* and *OOHDM*, *UWE* also explicitly identifies steps that can be performed in an automatic way. Moreover, it proposes a rich formalism for designing internal node content with windows, framesets and frames, a feature which *MESH* currently lacks. On the other hand, none of these UML based approaches allows for link subtyping and multiple specializations over both node and link types in the way *MESH* does. Nor do they propose a navigation paradigm that can be compared to *MESH*’s context based navigation. Yet another approach to modeling web applications with UML is [15]. However, the latter is restricted

to modeling the components and client/server behavior of HTML pages and barely discusses navigational issues.

MESH's context-based navigation paradigm tackles the disorientation problem by providing dynamic guided tours throughout the information space. *EORM*, *RMM*, *HDM*, *OOHDM* and *WebML* also feature specific topologies such as *guided tours*, *indexes* etc. A fundamental difference is that these are conceived as explicit *design components*, requiring author input for query definitions, node collections and forward/backward links. For instance in *OOHDM*, topologies are defined through *navigational contexts*. Several types of navigational contexts can be designed. However, upon implementation, they are to be authored semi-manually by means of explicitly stored queries or enumerations of nodes. Such queries are generated on-the-fly by *MESH*'s hyperbase engine, with the context node and context link type as input parameters. The author simply has to associate a link type with a user-interface event. No additional constructs are to be defined nor maintained. Moreover, such association can be made on node *type* level, hence has to be authored only once for a whole class of nodes. For instance a *painted-by* anchor needs to be defined only once for the entire node type **painting**, instead of for each **painting** instance. *MESH* has the guided tour as default construct for navigating through a set of nodes as generated after selection of a non-unique link type. Obviously, all kinds of indexes can be generated just as easily from the information in the linkbase/repository. Note also the importance of the ready availability of semantic metadata in the linkbase/repository. For example when a user wants to refine a guided tour, this can be accomplished by selecting a *child link type* from the original context link type, hence reducing the tour to nodes with a more specific semantic relationship to the context node.

RMM, *HDM* and *OOHDM* also in one way or another incorporate node visualization mechanisms that are sensitive to from where a node is accessed, as an alternative to traditional destination anchors. *RMM* divides a given entity type into *slices*, each one grouping a different set of *attributes*. Context-dependent visualization is implemented by denoting a "head" slice accessed by default, but allowing each link to target its own "destination slice". *HDM* and *WebML* have a very similar approach. In *OOHDM*, context-sensitive visualization is achieved by means of *context classes*, which are groupings of attributes that are only relevant to (a) specific context(s). They are only presented within the

corresponding context and determine the information (e.g. why a node belongs to a given guided tour) and also the source anchors (e.g. for moving forward and backward in a guided tour) to be shown according to this context. *OOHDM* also allows for contexts to be *nested*, such that complex, hierarchical navigation patterns can be constructed.

MESH, however, is the only approach to use a link's *type* as the parameter to determine the destination node's visualization. It is also the only approach that stays true to the object-oriented paradigm by looking upon destination anchors as *visualization routines* that provide an interface to a destination node, but hide their actual implementation from the source node. It is also the only approach where navigational actions are defined on *type level*, independently of actual link instances, such that both source and destination "anchors" can be defined to entire node types. Also, the end user is able to issue the same navigational actions to all nodes of a given tour at once, instead of for each separate node, as was discussed in section 2.4.3. Furthermore, these type-level navigational actions provide *MESH* with the unique ability to bookmark *a complete navigational situation* (instead of single nodes) in a very compact manner, with the possibility to resume navigation later on, from the exact point where it was left.

Finally, *MESH* has in common with *query-based* systems such as *Strudel* [20], [21] that web sites are defined by means of *database techniques*. Management of hypermedia structure and of "internal" node data are perceived as two orthogonal tasks. *Strudel* uses and extends a declarative query-language, *StruQL*, for website implementation, which explicitly allows for expressing integrity constraints. Another similarity is that "links" are modeled by means of queries. However, again, such queries are to be authored explicitly, in contrast to *MESH* where they are generated automatically at runtime, according to the user's actions. A database-like approach similar to *Strudel* can be found in *Araneus* [38]. As in *MESH*, pages are defined as instances of a page scheme. However, *Araneus* lacks *MESH*'s subtyping and inheritance mechanism.

4.3 Set based hypermedia

Set-based hypermedia paradigms such as *CHM* [19] and *Hyper-G/Hyperwave* [1], [30] equally provide inherent support for navigation in two orthogonal planes; *inside a collection* and *across collection boundaries*. Their *current container* and *current member* concepts are comparable to *MESH*'s *current context* and *current node* respectively. Also, they proclaim nodes to be self-contained entities, much like the object-oriented encapsulation principle, and even provide a measure of context-sensitive visualization. Furthermore, *Hyperwave* shares with *MESH* its ability to separate document content from structural links: the link structure is stored in a database, whereas document content is stored separately. As a consequence, it becomes very easy to maintain link integrity and to provide run-time generated overviews of the available information.

However, although some offer limited support for tagging documents with metadata such as *author*, *topic* etc., they lack the abstractions of a firm underlying conceptual data model with typed nodes and node interrelations. Likewise, the opportunity of issuing abstract navigational actions on tour level is a feature that is exclusive to *MESH*. On the other hand, research upon the set paradigm has certainly resulted in some interesting and fruitful insights in hypermedia development. Concepts such as *self-containment*, *context-dependent visualization* and *navigational context* originate (at least partially) in set-based systems. However, whereas these concepts were initially claimed to be particular to a set-based approach, *MESH*, among others, has proven that they can be successfully transferred to the node/link paradigm.

4.4 Adaptive hypermedia

Adaptive hypermedia systems [10], [54] allow for both node visualization and accessible link anchors to be influenced by an inference mechanism, based on knowledge about the *current user*. This knowledge is (partially) obtained by observing the user's previous actions. For instance direct guidance systems such as *Webwatcher* [29] suggest relevant links based on experience with previous users' browsing behavior and a set of keywords that has been provided at the beginning of the current user's session. One is "guided" along potentially relevant pages, with the system actually taking over navigation control, at least partially.

While featuring dynamic node content as well as dynamic links, *MESH* intentionally differs from typical adaptive hypermedia systems in that it does not try to *classify* the current user nor does it account for an explicit *user model*. Also, observation of the user's navigational actions is not aimed at *influencing* his behavior: *MESH*'s dynamism is merely targeted at facilitating navigation and providing a structured perspective upon the information space. A guided tour of *indirect* links is invariably the result of a *direct* link type selection by the user: the initiative fully resides with the latter. Therefore, adaptive hypermedia techniques could be seen as a complement to *MESH*, rather than a substitute. Indeed, especially in large hyperbases, with an oversupply of outgoing link anchors for a given source node, it could be required to impose dynamic behavior upon these *direct* links as well. A user profile could then be applied to highlight the most relevant direct link (type) anchors and hide irrelevant ones. This would entail a relaxation of the assertion of complete navigational flexibility and the end user having autonomous control over navigation strategy, with certain links only becoming visible or accessible (or certain links being advertised more prominently than others) after a certain precondition is fulfilled, as described e.g. in [11]. Hence, whereas *MESH*'s current dynamism entails the generation of guided tours *in accordance with the user's actions*, the latter could be assisted by adaptive techniques *suggesting which action to take*. In this respect, *MESH*'s rich modeling abstractions with typed nodes, links and aspects, as well as its *context* notion, could provide valuable semantic information as additional input to existing adaptation techniques. Also, rules could be applied at the desired level of granularity: general rules at parent node type level and specific rules at child node type, aspect type or even instance level. In that case, the inheritance and overriding mechanism would have to be extended to accommodate for such adaptation rules.

5 Conclusions and future work

5.1 General advantages of *MESH*

Whereas the development advantages of object-oriented analysis and design are too numerous to mention them all, a few topics can be named that particularly apply to the *MESH* approach. This first subsection briefly summarizes the *general* advantages of the *MESH* framework with respect to both end user orientation and maintainability. A more thorough evaluation is provided in [34]. A subsequent

subsection further discusses the advantages that result particularly from *MESH*'s relying on encapsulation and information hiding.

Obviously, the notion of *abstraction* is strikingly present in *all* components of the framework. Both *layout templates* and *link types* can be designed on a high level of generality, and refined and enriched on more concrete levels through inheritance and overriding. This not only facilitates authoring, but also benefits consistency, hence the overall quality of the application. The practice of attributing link types to node types, rather than just attributing links to individual nodes, along with the ability of enforcing *constraints* regarding a link type's domain, cardinalities and inverse, allows for checking on consistency and referential integrity. Such constraint preservation should not necessarily be "repressive", but may well be "preventive", by suggesting mandatory links and feasible destination nodes for a newly defined link instance, based on conceptual metadata stored in the linkbase/repository such as type information for nodes, aspects and links and the corresponding constraints. Moreover, the use of *higher-order* information units, i.e. node types and link types but also the representation of entire guided tours as **node.link-type** combinations decreases cognitive overhead, hence facilitates end user orientation. This is especially true because conceptual similarities between information units are also reflected in their visual appearance. Obviously, easy accessibility of meta information at runtime is also a prerequisite for the *context-based navigation paradigm*.

Indeed, another benefit of abstraction lies in *MESH*'s navigation paradigm, where navigational actions are specified on an abstract level, resulting in link *type* selection taking the place of link *instance* selection. This not only facilitates the *design*, with such actions being specified on node/aspect *type* level, but also yields node implementations with anchors that are *independent of the actual link instances*. Obviously, the context-based navigation mechanism also positively affects orientation by providing linear guidance to the user and a general sense of "context". Again, this effect is enhanced by the context influencing the visual appearance of the information units, i.e. a node can present that subset of its embedded information that is most relevant to the current context.

Finally, *visualizing the hypertext structure* is also beneficial to reducing cognitive overhead. The latter is facilitated by two factors; first, the hyperbase structure being stored within a *searchable* relational database, such that (partial) maps and overviews can be generated at run-time and the abundance of meta-information as node, aspect and link types, which allows for incorporating these abstractions into the overview. Consequently, maps are able to contain concepts of varying granularity, as applied in e.g. *fish-eye views*.

5.2 Advantages that result from the encapsulation and information hiding approach

The advantages that result particularly from *MESH*'s relying on encapsulation and information hiding are summarized below:

- Nodes interact by means of well-defined *public interfaces*, based on their set of attributed link types, but they can stay unaware of one another's actual implementation.
- This results in a very loose coupling between nodes, such that inter-node maintenance (i.e. updates to the link structure) can be executed orthogonally to intra-node maintenance (i.e. updates to node content). 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.
- Because a node encapsulates all behavior necessary for its own visualization, *node content* can be designed or updated and tested independently, without prior knowledge of the entire hypermedia structure nor related nodes' content. Multimedia objects are hidden in the node's implementation and are never to be referenced directly from the outside world. The only criterion for node design is the public interface defined by its attributed link types.
- The hypermedia system's *link structure* can be tested and navigated through even before the actual *node content* is implemented: it is solely based on relational database data in the *linkbase/repository*. Also, links can be reallocated without any modification to the source or destination node.
- Indeed, generic "source anchors" can be defined on an aggregate level in a node type's (or aspect type's) layout template by associating a user interface event with a link *type* selection, independently of the actual node and link instances. At runtime, the relevant link instance(s)

is/are distilled automatically from this type-level anchor, resulting in a single destination node access or a guided tour being generated 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.

- The *context-sensitive visualization* mechanism, as an alternative to destination anchors, 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. Also, visualization properties can 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. The proposed approach can be considered as more natural to traditional anchors in that a node does not react to *from which node* it is accessed, but to the *reason why*. Where necessary, general visualization routines can be overridden to provide for a more specific reaction by means of the *link subtyping* mechanism.
- Finally, the very loose definition of the node concept allows for an open system where documents/components of almost any type can be used as nodes and be seamlessly integrated into the system, while retaining full navigational flexibility. The heterogeneous nodes can interact effortlessly, by means of the well-defined public interfaces and hidden implementations. Where necessary, the hyperbase engine can generate a supplementary *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.

5.3 Current web-based prototype and future work

MESH's current prototype implementation provides a servlet-based hypermedia engine, which processes navigational actions and accesses a relational DBMS containing the linkbase/repository. The servlet also keeps track of run-time information such as the current context, which is however duplicated in persistent storage for robustness. Node *content* is presented as HTML code in a “traditional” webbrowser. However, because the framework leaves much freedom as to the actual

implementation of nodes and aspects, the HTML code can be stored as a single *static* file, it can be *assembled* from multiple HTML fragments, or it can be generated entirely *dynamically* at runtime. The only requirement is that it can be uniquely identified, e.g. by means of a URI. The webbrowser is enhanced with a “navigation applet”, which provides enriched user interface functionality and ad-hoc node overviews (although the applet is not strictly required for the core functionality of the system).

Navigational actions can be selected from either *within* an HTML document or from an applet-generated node overview. However, the HTML code does not contain direct references to related nodes. It only defines a node’s *multimedia content*: the *links* are stored outside the node’s content in the linkbase/repository, along with the conceptual metadata. A source anchor as represented in the HTML code consists of a reference to the *servlet’s URL and a parameter*. The latter identifies the direct link type or next/previous action to be selected by the anchor. Upon stimulation of the anchor, the parameter is passed to the servlet, which calculates the correct destination node and detects whether the action induces a context change. In the latter case, the corresponding indirect links are generated by the servlet, based on the selected link type and the information in the linkbase/repository. Hence the servlet’s support for navigational actions entails both within-tour navigation and the initiation of new guided tours, as required by the navigation paradigm.

Context-sensitive node visualization again depends on the origin of the HTML code: as to static HTML documents, it consists of defining a mapping between link type and HTML anchor, such that the destination document scrolls to the appropriate section when this link type is selected. As to HTML that is generated entirely dynamically, the link type is mapped to a node’s *presentation routine*, which in fact generates the HTML code for the node and its associated aspects. Obviously, the former is a rather rudimentary approach to context-sensitivity, whereas the latter allows for much more subtlety in adjusting a node’s visualization to the current context. An intermediate solution is the presentation routine dynamically assembling a single HTML document from different static HTML or XML *fragments*, representing the node’s and associated aspects’ content.

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 users with the right privileges to reallocate links and update node state and content during a navigation session. For that purpose, *MESH* will have to be extended to model *application behavior*. Similarly to *W2000*, *UHDM* and *UWE*, UML will be used for that purpose. This will entail the definition of UML extensions to be able to fully represent *MESH*'s modeling constructs. A related issue is the support for different user types, possibly with a different view on the nodes' content. For that purpose, a node's visualization routines should not only take the selected link type into account, but also the type of the current user. This feature is not yet standardized in *MESH*.

Another future research topic is the specification of richer layout templates for defining node or aspect content on an abstract level. At present, simple HTML is used for that purpose. A future version is to specify all of *MESH*'s node *instance* content by means of XML and all of its node *type* level templates by means of the XML Schema [48] specification. This in contrast to e.g. *WebML*, where XML itself is used to represent type-level information. Note, however, that *MESH*'s data model and navigation paradigm are completely independent of the actual node content, i.e. the entire network of nodes and links can be tested and navigated through based on the data and metadata in the linkbase/repository, before actually implementing the nodes' content.

Regarding node content, a related topic is the possibility to incorporate "borrowed" multimedia content from adjacent nodes, e.g. a **painter** node including pictures of his most significant **paintings**. One of the merits of *OOHDM* in this respect is that, in the navigational scheme, nodes as instances of navigational classes may combine attributes of different related conceptual classes. This is also true for *pages* as defined in *WebML*'s *Composition Model*. In *MESH*'s current state, a node can only "borrow" a single property of related nodes, namely their *description field*. The latter is stored within the database and hence can be easily retrieved and used in a given node, possibly as an outgoing anchor denoting a link to the former node. However, it would be more than desirable to provide a construct for allowing one node to present multimedia data that belongs to (and is maintained by) another node. Such construct could be seen as a node's template delegating a placeholder not to an aspect but to

another node that implements it. In the current implementation, these data has to be duplicated in each node it is visualized in. A similar extension to *RMM* is suggested in [28], with *m-slices* combining attributes belonging to different entities to be presented in a single window. An advantage in comparison to the *OOHDM* approach is that the content of each attribute is still retraceable to a single “owner” entity, which manages the attribute’s content. The practice of nodes containing brief information belonging to other nodes, e.g. including **painter** data in a **painting** node, is also applied to a *WWW* environment in [8], where constructs such as *transclusions* and *hot links* are suggested.

Finally, the current implementation restricts applicability of the navigation paradigm to a single web site or at the very least to multiple sites controlled by a single linkbase/repository. Therefore, future development efforts target a *distributed* linkbase/repository. Note, however, that the present restriction in scope is merely a consequence of the actual implementation, not of the navigation paradigm in se.

References

- [1] K. Andrews, F. Kappe and H. Maurer, The Hyper-G Network Information System, *Journal of Universal Computer Science*, Vol. 1, No. 4 (1995) 206-220.
- [2] H. Ashman, A. Garrido and H. Oinas-Kukkonen, Hand-made and Computed Links, Precomputed and Dynamic Links, *Proceedings of Hypertext - Information Retrieval - Multimedia (HIM '97)*, Dortmund (1997) 191-208.
- [3] J. Ayers et al., Synchronized Multimedia Integration Language (SMIL) 2.0, *W3C Recommendation* (2001).
- [4] A. Bapat, J. Wäsch, K. Aberer and J. Haake, An Extensible Object-Oriented Hypermedia Engine, *Proceedings of the seventh ACM Conference on Hypertext (Hypertext '96)*, Washington D.C. (1996) 203-214.
- [5] L. Baresi, F. Garzotto and P. Paolini, Extending UML for Modeling Web Applications, *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, Hawaii (2001) 1285-1294.

- [6] H. Baumeister, N. Koch and L. Mandel, Towards a UML extension for hypermedia design, Proceedings of UML '99 The Unified Modeling Language – Beyond the Standard, LNCS 1723, Fort Collins (1999) 614-629.
- [7] M. Bernstein, The Navigation Problem Reconsidered, Hypertext/Hypermedia Handbook, E. Berk and J. Devlin Eds., (McGraw-Hill, New York, 1991) 285-297.
- [8] M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian and H. Oinas-Kukkonen, Fourth generation hypermedia: some missing links for the World Wide Web, International Journal of Human-Computer Studies Vol. 47, No. 1 (1997) 31-65.
- [9] A. Botafogo, E. Rivlin, and B. Shneiderman, Structural analysis of hypertext: identifying hierarchies and useful metrics, ACM Trans. Inf. Syst. Vol. 10, No. 2 (1992) 142-180.
- [10] P. Brusilovsky, Methods and techniques of adaptive hypermedia, User Modeling and User-Adapted Interaction Vol. 6, No. 2-3 (1996) 87-129.
- [11] L. Calvi and P. De Bra, Improving the Usability of Hypertext Courseware through Adaptive Linking, Proceedings of the eighth ACM Conference on Hypertext (Hypertext '97), Southampton (1997) 224-225.
- [12] R. Cattell and D. Barry, The Object Database Standard: ODMG 3.0 (Morgan Kaufmann Publishers Inc., San Francisco, CA, 2000).
- [13] S. Ceri, P. Fraternali and S. Paraboschi, "Web Modeling Language", (WebML): a modeling language for designing Web sites. Proceedings of the 9th. International World Wide Web Conference (2000) 137-159.
- [14] A. Cockburn and S. Jones, Which way now? Analyzing and easing inadequacies in WWW navigation, International Journal of Human-Computer Studies No. 45 (1996) 105-129.
- [15] J. Conallen, Modeling Web Applications with UML, white paper (1999).
- [16] H. Davis, W. Hall, I. Heath, G. Hill and R. Wilkins, MICROCOSM: An Open Hypermedia Environment for Information Integration, Computer Science Technical Report CSTR 92-15 (1992).
- [17] H. Davis, To Embed or Not to Embed, Commun. ACM Vol. 38, No. 8 (1995) 108-109.
- [18] H. Davis, S. Reich and A. Rizk, OHP - Open Hypermedia Protocol, Working Draft 2.0 (1997).
- [19] E. Duval, H. Olivié and N. Scherbakov, Contained Hypermedia, Journal of Universal Computer Science, Vol. 1, No. 10 (1995) 687-705.

- [20] M. Fernández, D. Florescu, J. Kang, A. Levy and D. Suciu, Catching the boat with Strudel: experiences with a Web-site management system, *Proceedings of the ACM SIGMOD international conference on Management of data*, Seattle (1998) 414-425.
- [21] M. Fernández, D. Suciu and I. Tatarinov, Declarative specification of data-intensive Web sites, *Proceedings of the second conference on Domain-specific languages*, Austin, TX (1999).
- [22] R. Furuta and P. Stotts, The Trellis Hypertext Reference Model, *Proceedings of the Workshop on Hypertext Standardisation*, Special Publication SP500-178, National Institute of Standards and Technology, Gaithersburg (1990) 83-93.
- [23] F. Garzotto, P. Paolini and D. Schwabe, HDM - A Model-Based Approach to Hypertext Application Design, *ACM Trans. Inf. Syst.* Vol. 11, No. 1 (1993) 1-26.
- [24] F. Garzotto, L. Mainetti and P. Paolini, Hypermedia Design, Analysis, and Evaluation Issues, *Commun. ACM* Vol. 38, No. 8 (1995) 74-86.
- [25] A. Ginige, D. Lowe and J. Robertson, Hypermedia Authoring, *IEEE Multimedia* Vol. 2, No. 4 (1995) 24-35.
- [26] F. Halasz, Reflections on NoteCards: Seven Issues for Next Generation Hypermedia Systems, *Commun. ACM* Vol. 31, No. 7 (1988) 836-852.
- [27] F. Halasz and M. Schwartz, The Dexter hypertext reference model, *Commun. ACM* Vol. 37, No. 2 (1994) 30-39.
- [28] T. Isakowitz, A. Kamis and M. Koufaris, The Extended RMM Methodology for Web Publishing, Working Paper IS-98-18, Center for Research on Information Systems (1998).
- [29] T. Joachims, D. Freitag and T. Mitchell, Webwatcher: A Tour Guide for the World Wide Web, CMU research report (1996).
- [30] F. Kappe, Managing Knowledge with Hyperwave Information Server, Hyperwave White Paper Version 1.2 (1999).
- [31] T. Knopik A. and Bapat, The Role of Node and Link Types in Open Hypermedia Systems, *Proceedings of the sixth ACM European Conference on Hypermedia Technology (ECHT '94)*, Edinburgh (1994) 33-36.

- [32] N. Koch, A. Kraus and R. Hennicker, The Authoring Process of the UML-based Web Engineering Approach, Proceedings of the First International Workshop on Web-oriented Software Technology (IWWOST'01), Valencia (2001).
- [33] D. Lange, An Object-Oriented design method for hypermedia information systems, Proceedings of the twenty-seventh Hawaii International Conference on System Sciences (HICSS-27), Hawaii (1994) 366-375.
- [34] W. Lemahieu, Improved Navigation and Maintenance through an Object-Oriented Approach to Hypermedia Modeling, Doctoral dissertation, Department of Applied Economic Sciences, Katholieke Universiteit Leuven, 1999.
- [35] W. Lemahieu, MESH: A Model-Based Approach to Hypermedia Design, in: Chen, Q. (ed.) Human Computer Interaction: Issues and Challenges (Idea Group Publishing, Hershey, PA, 2001) 167-184.
- [36] W. Lemahieu, Context-based navigation in the Web by means of dynamically generated guided tours, Computer Networks Journal 39(3) (2002) 311-328.
- [37] M. Mayes, A method for evaluating the efficiency of presenting information in a hypermedia environment, Computer in Education Vol. 18, No. 1 (1994) 179-182.
- [38] G. Mecca, P. Atzeni, A. Masci, P. Merialdo and G. Sindoni, The Araneus Web-Base Management System, Exhibits Program of ACM SIGMOD (1998) 544-546.
- [39] B. Meyer, Object-Oriented Software Construction, Second Edition (Prentice Hall Professional Technical Reference, Santa Barbara, 1997).
- [40] N. Meyrowitz, 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 (1986) 186-201.
- [41] D. Millard, S. Reich and H. Davis, Reworking OHP: the road to OHP-Nav. Proceedings of the fourth Workshop on Open Hypermedia Systems at Hypertext '98, Pittsburgh (1998) 48-53.
- [42] J. Nanard and M. Nanard, Hypertext Design Environments and the Hypertext Design Process, Commun. ACM Vol. 38, No. 8 (1995) 49-56.
- [43] C. Ramaiah, An Overview of Hypertext and Hypermedia, International Information, Communication & Education Vol. 11, No. 1 (1992) 26-42.

- [44] G. Rossi, D. Schwabe and F. Lyardet, Web application models are more than conceptual models, Proceedings of the World Wild Web and Conceptual Modeling'99 Workshop, ER'99 Conference, Paris (1999) 239-252.
- [45] G. Rossi, D. Schwabe and F. Lyardet, Abstraction and Reuse Mechanisms in Web Application Models, Proceedings of the World Wild Web and Conceptual Modeling'00 Workshop, ER'00 Conference, Salt Lake City (2000) 19-21.
- [46] H. Schutt and N. Streitz, HyperBase: A Hypermedia Engine Based on a Relational Data Base Management System, Proceedings of the European Conference on Hypertext, Versailles (1990) 95-108.
- [47] M. Snoeck, G. Dedene, M. Verhelst and A. Depuydt, Object-Oriented Enterprise modeling with MERODE (Universitaire Pers Leuven, Leuven, 1999).
- [48] H. Thompson, D. Beech, M. Maloney and N. Mendelsohn Eds., XML Schema Part 1: Structures, W3C Recommendation (2001).
- [49] M. Thüring, J. Haake, and J. Hannemann, 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 (1991) 161-177.
- [50] M. Thüring, J. Hannemann and J. Haake, Hypermedia and Cognition: Designing for comprehension, Commun. ACM Vol. 38, No. 8 (1995) 57-66.
- [51] R. Trigg, Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment, ACM Trans. Office Inf. Syst. Vol. 6, No. 4 (1988) 398-414.
- [52] R. Wieringa, Steps towards a method for the formal modelling of dynamic objects, Data & Knowledge Engineering Vol. 6 (1991) 509-540.
- [53] U. Wiil and J. Leggett, Hyperform: a hypermedia system development environment, ACM Trans. Inf. Syst. Vol. 15, No. 1 (1997) 1-31.
- [54] H. Wu, E. De Kort and P. De Bra, Design Issues for General-Purpose Adaptive Hypermedia Systems. Proceedings of the ACM Conference on Hypertext and Hypermedia, Aarhus, Denmark (2001) 141-150.

Author information

Wilfried Lemahieu holds a Ph. D. from the Department of Applied Economic Sciences of the Katholieke Universiteit Leuven, Belgium (1999). At present, he is a full professor at the Management Informatics research group of this department. His teaching includes Database Management, Data Storage Architectures and Management Informatics. Besides hypermedia systems, his research interests comprise object-relational and object-oriented database systems, distributed object architectures and web services.

