Composites in a Dexter-Based Hypermedia Framework

Kaj Grønbæk
Computer Science Department,
Aarhus University,
Ny Munkegade 116, Bldg. 540
DK-8000 Århus C, Denmark.
fax: +45 8942 3255
e-mail: kgronbak@daimi.aau.dk

ABSTRACT

This paper discusses the design and use of a generic composite mechanism in the object oriented DEVISE Hypermedia (DHM) development framework. The DHM framework is based on the Dexter Hypertext Reference Model, which introduces a notion of composite to model editors with complex or multiple types of contents. The original Dexter notion of composites is, however, insufficient to cover structural composites including or referencing other components. Thus the DHM framework has been extended with generic composite classes suited to support structures within the hypermedia network itself. The paper presents and discusses the design of the generic composite classes belonging to the STORAGE and RUNTIME layers of the framework. A central aspect of the design is that the structuring mechanism is a true composite with a collection of components as its contents rather than an atomic component with links to other components as in the classical systems such as NoteCards, Intermedia, and KMS. It is also shown how the powerful generic classes can be used to implement a variety of useful hypermedia concepts such as: hierarchy by inclusion, hierarchy by reference, virtual and computed browsers, TableTops and GuidedTours.

KEYWORDS: Composites, Structure, Hierarchies, GuidedTour, Dexter model, Object Oriented Framework.

1 INTRODUCTION

This paper discusses design and uses of composites in a Dexter based hypermedia development framework, the DEVISE Hypermedia (DHM) framework. The DHM framework is a generic object oriented framework for developing hypermedia systems that are compliant to the Dexter Hypertext Reference Model [9, 10]. Issues in the design of the DHM framework and applications have been discussed in earlier papers [5, 8]. Grønbæk & Trigg [8] covers some initial discussion on the notion of composites in the DHM framework. This paper delves into more details on the object oriented design of the generic and powerful composite concept in DHM, it also provides a discussion on applications of the generic concept. One of the major applications of the DHM framework is to be part of a *CSCW Open Development Environment* (Esprit III project EuroCODE), hence the paper among other applications discusses how composite structures can be used to support communication between hypermedia authors. This involves a re-implementation of Trigg's [18] notions of TableTops and GuidedTours by means of the generic Composite concept of DHM.

The Dexter Model

The *Dexter Hypertext Reference Model* [9, 10] (called "Dexter" in the rest of this paper) separates a hypertext system into three layers. The *Storage layer* captures the persistent, storable objects making up a *hypertext* which consists of a set of components. *Component* is the basic object provided in the Storage layer. The component includes a *contents* specification, a general purpose set of *attributes*, a *presentation specification* ("PSpec"), and a set of *anchors*. The *atomic component* is an abstraction replacing the widely used but weakly defined concept of 'node' in a hypertext. *Composite components* provide a hierarchical structuring mechanism. The content of a *link component* is a list of *specifiers*, each including a presentation specification as well as component and anchor identifiers. The *Within-component layer* corresponds to the data objects, the contents of components, and the individual editors to handle the data objects. The *Runtime layer* is responsible for handling links, anchors, and components at runtime. Objects in the runtime layer include *Session*, managing interaction with a particular hypertext; and *Instantiation*, managing interaction with a particular component. The

runtime layer provides editor independent user interface facilities.

This paper assumes a basic knowledge of the Dexter model. Such knowledge can be achieved from [8, 10] appearing in Communications of the ACM, February 1994, which includes a special section on Dexter based hypermedia.

Framework and Applications

The DHM framework is developed in the Mjølner BETA System (MBS), see [15, 16]. It applies object oriented database technologies for storing the objects corresponding to the Dexter Storage layer, see [3, 5, 14]. The DHM framework will work on any hardware platform where the Mjølner BETA System is implemented. Currently this includes a variety of UNIX/X-windows platforms (SUN OS 4.1.x, SUN Solaris, HP UX7/8), Macintosh OS (System 7), and by August 1994 the PC/Windows platform. Several variants of working prototypes (called DEVISE Hypermedia, or just "DHM") have been developed for the UNIX and Macintosh platforms. The variants include single user and multi-user implementations, and a cross platform version is also implemented running an OODB server on a UNIX machine and clients running simultaneously on UNIX hosts and Macs. Finally, an experimental prototype has been developed embedding a recently developed interpreter for the BETA language to obtain a runtime tailorable hypermedia system, see [6].

Structure of the Paper

The structure of the paper is as follows. Section 2 gives a brief account of the composite concept in previous literature. Section 3 discusses the design of the generic composite classes provided by the DHM framework, and the subsequent sections discuss uses of these classes. Section 4 discusses different kinds of hierarchical structures. Section 5 discusses how to provide virtual computed composites for browsers and queries. Section 6 discusses how to support communication of hypermedia structures by means of composites. Section 7 concludes the paper.

2 THE COMPOSITE CONCEPT

Different notions of structure have been implemented in most hypermedia systems. In many hypermedia systems, e.g. KMS [2], Intermedia [20], and NoteCards [12] the implemented link facility is also used to support different kinds of structures that are not inherently "references" or "associations" as links originally were aimed at handling.

Examples of such structures are the frame hierarchies of KMS, which are built by means of adding link properties to so-called *tree items* that are hard to distinguish semantically from *annotation items* used for non-hierarchical cross-references. Similarly, FileBox cards in NoteCards are aimed at handling hierarchical structures by providing a special FiledCard link type to point out the cards contained in a FileBox. Other examples are collections of "hits" by query searches. KMS, Intermedia and NoteCards altogether treat these by making an atomic "result" node with links to all the "hits". Finally, TableTops and GuidedTours in NoteCards are implemented by having special link types (TableTop Link, GuidedTour Link) pointing from special atomic nodes (called TableTop Card and GuidedTour Card) to the nodes being conceptually "contained". Using links for such structures typically implies that the hypermedia network contains many system generated links that users have to be able to distinguish from the links they created explicitly. Such solutions for structuring appear to be cumbersome and inflexible to handle.

In his "Seven Issues" paper, Halasz [11] criticized purely link-based structures arguing that they lack a single node capturing the overall structure. Accordingly he proposed *composites* to become first class citizens in hypermedia together with atomic nodes and links. Composites would provide means of capturing non-link based organization of information, making structuring beyond pure link networks an explicit part of hypertext functionality.

Halasz also proposed the related notions of *computed* and *virtual* composites. The contents of a computed composite could be the result of a structural query over the hypertext returning sets of nodes and links as "hits." A virtual composite is created on demand at runtime, but not saved in the database.

The Dexter model's composite [9, 10] addresses this call for a non-link based structuring mechanism. As a collection of base components, it acts both as a full-fledged node in the network, and as a container for the structured data.

Though Dexter's notion of composite is a significant step forward, it is only one point in a spectrum of possible designs, each having certain advantages and meeting certain needs. Grønbæk & Trigg [8] discuss the limitations of the Dexter model's Composite and identify several other types of useful composites. In short, the original Dexter Composite is only aimed at containing collections of BaseComponents, which is the concept to represent the actual data contents of a component. This means that it is well suited to represent single nodes containing different types of data contents. However, to handle structures within the hypermedia network itself, composite types are needed that can contain or reference other components. Similar needs are also proposed by Hardman et al. [13], who discusses the usage of Dexter composites for representing multimedia data in hypermedia structures.

Composites Representing Hypermedia Sructures

Grønbæk & Trigg [8] discuss several examples of such composite types among those are LinkComposites to hold a collection of LinkComponents and AtomComposites to hold a collection of AtomComponents. They summarize their discussion in a table depicting three independent dimensions along which a hypermedia designer can choose when designing composites (and components in general) using the DHM framework. Having revisited these three

Structure	Type	Definition	Location
• Unstructured collection • Structured collection: - sorted list - keyed table - tree - ...	• Data objects • Components - restricted types - unrestricted	• Encapsulated in this composite • Globally visible	• within composite (inclusion) • outside composite (reference)

Table 1: Four aspects of composite contents.

dimensions for the design of new composite types it appears that the distinction on location of contents also applies to the component type of contents, hence the choices are better described as four dimensions. This is to be able to make distinctions between *reference* relations and *inclusion* relations as proposed by Halasz [11]. The modified table with four dimensions of choices for composite's contents is given in Table 1.

In [8] examples of several composites that span the original three dimensional space of choices are given. An example of a composite where the distinction of the fourth *Location* dimension applies is the so-called ContainerComposite that will be discussed further in Section 4.2. It is used to make structures similar to file system directory trees for organizing components. Compared to, e.g. the AtomComposite introduced in [8], a ContainerComposite's content is located within the composite and is only accessible through the composite. Note that the *Location* dimension describes options for instances whereas the *Definition* dimension describes options for placement of class definitions.

3 COMPOSITE CLASSES IN THE DHM FRAMEWORK

This section describes and discusses the generic composite classes in the DHM framework. First, the classes belonging to the STORAGE layer are discussed. These classes define the schema for the objects that are stored in the object oriented database[5]. Second, the classes belonging to the RUNTIME layer are discussed. These classes define the generic behavior of the composites when users interact with them at runtime.

3.1 STORAGE Layer Classes

The main class in the Storage layer is the *Hypertext*. A hypertext encapsulates the set of components that is stored and retrieved from the Persistent Store/OODB. The hypertext class has the *Component* class as a main nested class. The component class has a variety of subclasses including *AtomComponent*, *LinkComponent* and *CompositeComponent* that corresponds to the similar Dexter model concepts. In the DHM framework, the AtomComponent and CompositeComponent concepts have been classified as subclasses of an abstract superclass called NonLinkComponent. This is convenient in situations where one want to treat all atomic and composite components as a whole ex-

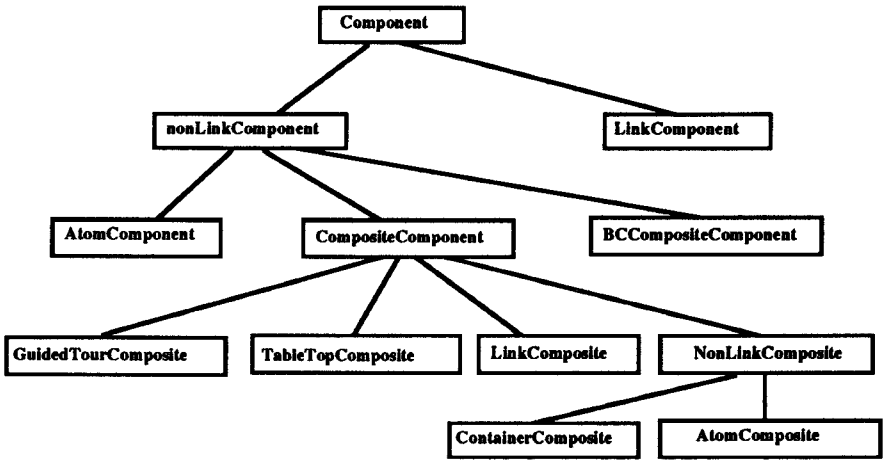

Figure 1: Inheritance hierarchy for the generic Component classes of the STORAGE Layer.

cluding links. An example of this is given later in Section 5.1. The Component inheritance hierarchy is depicted in Figure 1.

The AtomComponent class is aimed at handling "simple" data objects, such as text, and the LinkComponent class represents links in the framework. These classes are not discussed any further in this paper.

```
compositeComponent: Class nonLinkComponent
(#
     baseComponentType bind composite;
     ...
     (* Determine whether this composite should
        be treated as virtual. *)
     isVirtual: boolean

     (* Add a component to this composite *)
     addComponent: ...

     (* Remove a component *)
     removeComponent: ...

     (* Clear the contents of this composite *)
     clearComponents: ...

     (* Removes pointers to all components
        marked as 'deleted'.*)
     cleanup: ...

     (* Performs an action for each
        component in the composite. *)
     scanComponents: ...

     ...
#);
```

Table 2: Excerpt of the interface for the generic CompositeComponent class, which in turn inherits from the generic Component and NonLinkComponent classes.

The BCCompositeComponent corresponds to the original Dexter model composite and it consists of a set of BaseComponents, thus the prefix 'BC'. The BCCompositeComponent is aimed at modelling editors with multiple types of data objects, e.g. a form with fields with different types of anchorable contents. A BaseComponent models a type of data object directly. The BCCompositeComponent has procedures to add, remove, scan and access the BaseComponents of its contents.

The CompositeComponent class is aimed at handling collections of other components, and it is a new construction compared to the original Dexter model. The CompositeComponent also possesses procedures to access the components contained in the base (see Table 2). The value of the isVirtual Boolean attribute determines whether the composite is virtual or not. The LinkComposite is a specialized CompositeComponent that is restricted to contain LinkComponents. Similarly, a NonLinkComposite is a CompositeComponent restricted to contain only NonLinkComponents, and an AtomComposite is a NonLinkComposite further restricted to contain only AtomComponents. Specific uses of these classes are discussed in Sections 4-6.

Composites Implemented without Links

Classical systems like NoteCards [12], KMS [2] and Intermedia [20] implement composite-like structures such as hierarchies, query result nodes, and browsers by means of creating links between the special atomic node and its related nodes. For example in the case of computing a query or a browser, the system implicitly generates many links that only serve the purpose of implementing the node-"composite" relation.

In DHM, we have taken another approach and implemented the relationship between composites and their member components by means of pointers to objects. This makes membership of a composite a one-way relation, that is, components do not know about which and how many composites they are members of. An exception is the ContainerComposite as discussed in Section 4.2. An advantage of this one-way pointer solution is that removal of a composite can be done in constant time. This is in contrast to, e.g. the implementation of browsers in NoteCards. In NoteCards, deletion of a browser requires that all the nodes being a member of the composite should be visited to delete the back-link information. Another advantage is that components can be added and removed from composites without requiring change of locking on the member components in a cooperative use setting [5]. Adding links to a component requires linking access, to the component, because an anchor has to be created, to support bi-directionality.

A potential disadvantage with one-way relations is to keep composites updated when member components are deleted. By implementing composite-member relationships by means of bi-directional links, it is easy to update composites when members are deleted. This is more difficult when dealing with one-way relations, but in DHM a lazy approach similar to the handling of links to deleted components is chosen. When a component is deleted, it is removed from the hypertext, its data is destroyed, and a deleted 'flag' is set on it. Following a link to such deleted components results in a dangling link exception, see [8]. Similarly, deleted components that are members of composites are shown with a special 'deleted' icon when the composite is presented at runtime, and the user can remove it from the composite manually, or a cleanup operation can remove all in one operation. When the last pointer to a deleted component is removed, then the garbage collector reclaims it. The cleanup operation can be called for all *open* composite instantiations when changes occur in an arbitrary instantiation, this ensures consistent views. Calls to cleanup can also be performed upon presentation of a composite, ensuring that no "deleted" components are made visible to the user. The schema for cleanup should be determined by preferences set by the user.

3.2 RUNTIME Layer Classes

The main classes of the RUNTIME Layer are *Session* and *Instantiation*. A Session manages and manipulates a single Hypertext. (A hypertext can have several open sessions managed by the same or different users.) Runtime management of components is handled by the Instantiation class defined within the scope of the session class. (Components can have multiple open instantiations.) Runtime management of anchors is in the case of MarkedAnchors handled by *LinkMarkers*. In the case of WholeComponentAnchors and UnMarkedAnchors, there is no explicit Runtime object representing the anchor [8].

```
CompositeInst: Class Instantiation
(# componentType: bind compositeComponent;
     ...
     (* Present this instantiation. *)
     present: bind ...

     (* Determines whether the composite
        should be recomputed when presented.*)
     isComputed: boolean;

     (* Trash composite's current contents and
        recompute it.*)
     RecomputeComponent: bind ...

     (* Calls the Composite's addComponent *)
     addComponent: virtual ...

     (* Calls the Composite's removeComponent *)
     removeComponent: virtual ...

     (* Returns the selected component(s). *)
     SelectedComp:...

     Subinst: Class ... (* See Table #4 *)

     newsubInst: virtual ...

     (* Removes subInst for a specified comp *)
     removesubInstForComp: ...

     subInstList: ...

     subInstSelectedList: ...
     ...
#);
```

Table 3: Excerpt[1] of the interface for the generic CompositeInst class that inherits from the Instantiation class.

Figure 2 depicts the generic subclasses of the Instantiation class which are provided to handle different component types of the Storage Layer. Instantiations handling AtomComponents and LinkComponents are not discussed any further in this paper. The CompositeComponents have generic Instantiation subclasses to handle them at runtime.

The BCCompositeInst class has procedures for handling the *BaseComponents* contained in the corresponding BCCompositeComponent. Similarly, the CompositeInst (see Table 3) has a number of procedures for manipulating the set of *components* contained in the base to the corresponding CompositeComponent. Finally, it introduces a nested SubInst class for handling the contained components at Runtime without making real instantiations for them.

Light Weight Instantiations

To manage control of individual components in a composite without making a full-fledged Instantiation, a light weight instantiation class called SubInst (see Table 4) is introduced.

```
subInst: Class
(#
     componentType: bind thehypertext.component;

     theComponent: componentType;

     myInst: ...
     ...
     (* Called when a new subInst is created.*)
     init: virtual ...

     (* Present this subInst according to
        the display info.*)
     present: virtual ...

     (* Close down and remove the subInst.*)
     unpresent: virtual ...

     (* Save the display attributes associated
        this subinst.*)

     save: virtual ...

     (* Save and unpresent this subInst. *)
     close: virtual ...

     (* Hit, Selected and Deleted are virtuals
        being called whenever a corresponding
        event happens to the corresponding icon
        in the browser application.*)
     hit: virtual ...

     selected: virtual ...

     deleted: virtual ...

#); (* End of subInst class *)
```

Table 4: Excerpt from the generic SubInst class. This class is encapsulated in the CompositeInst class, described in Table 3.

The main difference to a real Instantiation is that a SubInst does *not* provide an application to handle the represented component and its data contents. It only provides an interface to access the component attributes and to invoke a real Instantiation for it. A SubInst typically appears in the user interface as an icon in a composite's editor window, e.g. the icon for 'Overview' in Figure 3.

[1] Among the things omitted here are attributes and procedures to handle multi-user behavior.

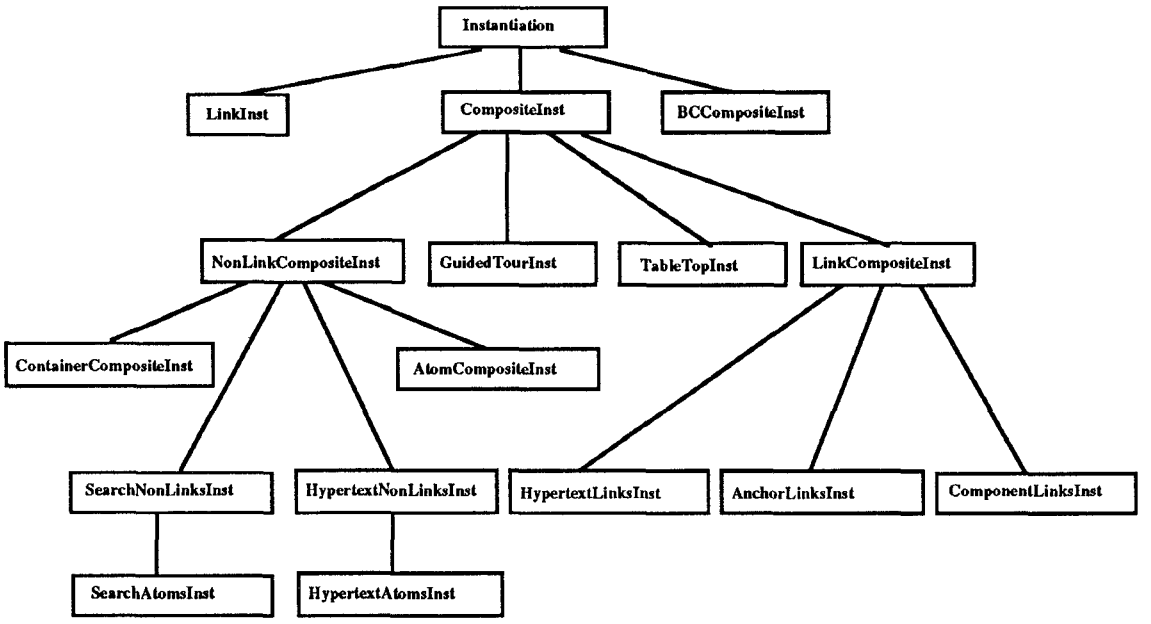

Figure 2: Inheritance hierarchy for the generic Instantiation classes of the RUNTIME Layer.

3.3 Virtual and Computed properties

Any composite can be made virtual by setting a flag. Such composites resemble normal composites, but they are usually not saved in the database. If however, a link or another composite reference a virtual composite, then it is indeed saved. Virtual composites resemble objects in a dynamic programming environment; if they are not referenced, then garbage collection reclaims them.

Similarly, any composite can be the result of a computation or it can be manually created by the user. A typical example of a computed composite is a composite created from executing a query. An attribute contains the information used to perform the computation. The composite's contents can later be re-computed, either on demand or automatically.

In [8] it was proposed to have virtualness and computedness as static properties of components. Hence special STORAGE layer classes like VirtualAtomComposite and VirtualLinkComposite were introduced, and objects instantiated from these classes remained virtual (and computed) forever. The *virtualness* of the VirtualAtomComposite means that it is not added to the hypertext's component list, i.e. it is only stored persistently if a link to it is created. The *computedness* implies that the query specification is stored in an attribute on the composite and if it is presented later it will recompute its contents from the query specification. However, this turned out to be inflexible for some use situations. Say a user has started making a collection of material by performing a query resulting in a VirtualAtomComposite referencing the set of AtomComponents matching the query. This is, however, inconvenient for our user who would like to continue picking some additional material "manually" and add it to the composite at hand finally making it into a static collection which is stored through the hypertext's component list as usual. Having faced this inconvenience, the DHM composite classes have been changed to have virtualness and computedness as dynamic properties not requiring subclassing. This means that our user can start from a query which by default produces a computed and virtual composite, which can dynamically be changed into an ordinary static and non-virtual user created composite.

3.4 Editing Composite Contents

Besides having composites with computed contents it is possible to manually edit the contents of a composite as mentioned above (see examples of composite user interface in Figure 3). Editing operations include: renaming components, repositioning icons, removing components, moving or copying references to components between composites, and "physical" moving of components between ContainerComposites, see Section 4.2 for more details about ContainerComposites. Copying and moving components between composites are in the user interface done in the Cut, Copy, and Paste paradigm. In the future Drag and Drop moving and copying between composites may be supported.

3.5 General Support for Partial Interchange

The generic Non-Link composite class of the DHM framework has been extended with a procedure to export an interchange format for the subset of a hypermedia network defined by the composite's contents. This means that a tagged ASCII text file following the SGML-like syntax proposed by the Dexter model is produced for the contained non-link components and the LinkComponents having endpoints in these components. Issues in managing partial interchange are discussed in a forthcoming paper [7].

4 COMPOSITES FOR HIERARCHICAL STRUCTURING

As mentioned in the introduction, many hypertext systems provide means for handling tree like hierarchical structuring. In DHM, such structuring is provided by means of different types of composite components. Halasz [11] proposes that when introducing composites to handle hierarchical structures, it should be possible to make distinctions between *reference* relation and *inclusion* relation. A reference relation between a composite and another component in our context means that the composite references the component that is physically located outside the composite itself. In contrast, an inclusion relation means that the composite *contains* the component such that, e.g. deletion of the composite implies deletion of the included component. Composites supporting inclusion relations can maintain hierarchies resembling a physical directory or folder structure. This distinction is expressed in the fourth column of Table 1 labelled 'Location'. Below examples of composites implementing hierarchy by references and hierarchy by inclusion are given.

4.1 Hierarchy by Reference

KMS [2] and NoteCards [12] provide hierarchical structures where each frame or card may be part of many hierarchies. In KMS, any frame can be referenced by an unlimited number of tree item link properties. In NoteCards, any card can be linked to by an unlimited number of FileBoxes. There is, however, also a weak notion of physical containment in NoteCards, a card must be referenced by at least one FileBox, i.e. there is an "Orphans" FileBox where cards that have been removed from all their FileBoxes go. In DHM, the generic composite classes CompositeComponent, NonLinkComposite, AtomComposite and LinkComposite provide such means of structuring. A component of a certain type can be referenced by any number of composites suited to reference the specific type of component. For example, an atomic TextComponent may be referenced by a CompositeComponent and an AtomComposite at the same time. Hence, these generic DHM composites can be used to create hierarchies similar to NoteCards FileBox and KMS TreeItem hierarchies. The generic classes can also be further specialized and for instance be restricted to only contain a certain type of components as leaves, e.g. in a document structuring context, a ChapterComposite could be restricted to contain or reference SectionComposites only, and in turn SectionComposites could be restricted to contain or reference AtomComponents.

Various CompositeInst class specializations can be developed to handle the behaviour for composites. Figure 2 depicts such generic classes, e.g. NonLinkCompositeInst and AtomCompositeInst. Several of the CompositeInst subclasses use the same CompositeComponent class.

4.2 Hierarchy by Inclusion

An object instantiated from the Hypertext class is in the original Dexter Model context a flat collection of components. Similar to Halasz' call for composites to handle inclusion, we have experienced that hypermedia network structures need a concept similar to directories in a file system. In DHM, we have introduced a composite type, the ContainerComposite, that behaves much like directories. The ContainerComposite is a specialization of the NonLinkComposite class, i.e. it can hold a mixed set of AtomComponents and other CompositeComponents and thereby other ContainerComposites. By means of ContainerComposites it is possible to maintain a directory tree like organization of a hypertext, where the hypertext itself becomes the root directory. For example, the SEPIA [17] hypermedia system also introduces a composite node to represent sub graphs in the network and folders in the various activity spaces.

The ContainerCompositeInst class handles the behaviour of ContainerComposites at runtime. Among other things, it implements "physical" movement of components between containers. As mentioned in Section 3.4 movement is performed by Cut and Paste procedures.

5 VIRTUAL AND COMPUTED COMPOSITES FOR BROWSERS AND QUERIES

The classical systems NoteCards [12] and Intermedia [20] provide browsers that are first class nodes in the systems. These browsers are typically implemented by means of common atomic nodes that maintain a set of links to the nodes represented in the browser. Similarly, results from query searches are represented as an atomic node having a set of system generated links to the "hits" of the query. Halasz [11] criticizes this approach and argues for introducing virtual composites which are transient structures that are dynamically computed at runtime. He also requires that such composites should behave as first class nodes in the network, i.e. they should be linkable and browseable as are any other nodes.

The DHM framework provides support for such virtual and computed composites at a general level. The CompositeComponent interface shown in Table 2 shows that an 'isVirtual' flag can be set on a composite (usually at creation time) to determine that this composite should not be stored persistently, i.e. it is not inserted in the hypertext's component list. Similarly, the general composite instantiation which interface is shown in Table 3 has a flag 'isComputed' to determine whether the composite's contents should be (re)computed at presentation. Moreover, the CompositeInst has a virtual procedure 'RecomputeComponent' that can be further bound in specializations to determine *how* the composite's contents should be computed. This procedure is invoked automatically by the framework when a composite is presented, but it may also be invoked on user request from the user interface, cf. the 'Recompute' button in Figure 3. All composites appear as first class linkable components independent of the setting of the 'isVirtual' and 'isComputed' flags. In addition, the status of a composite may be changed dynamically. If a user at some point decides that a given browser which was

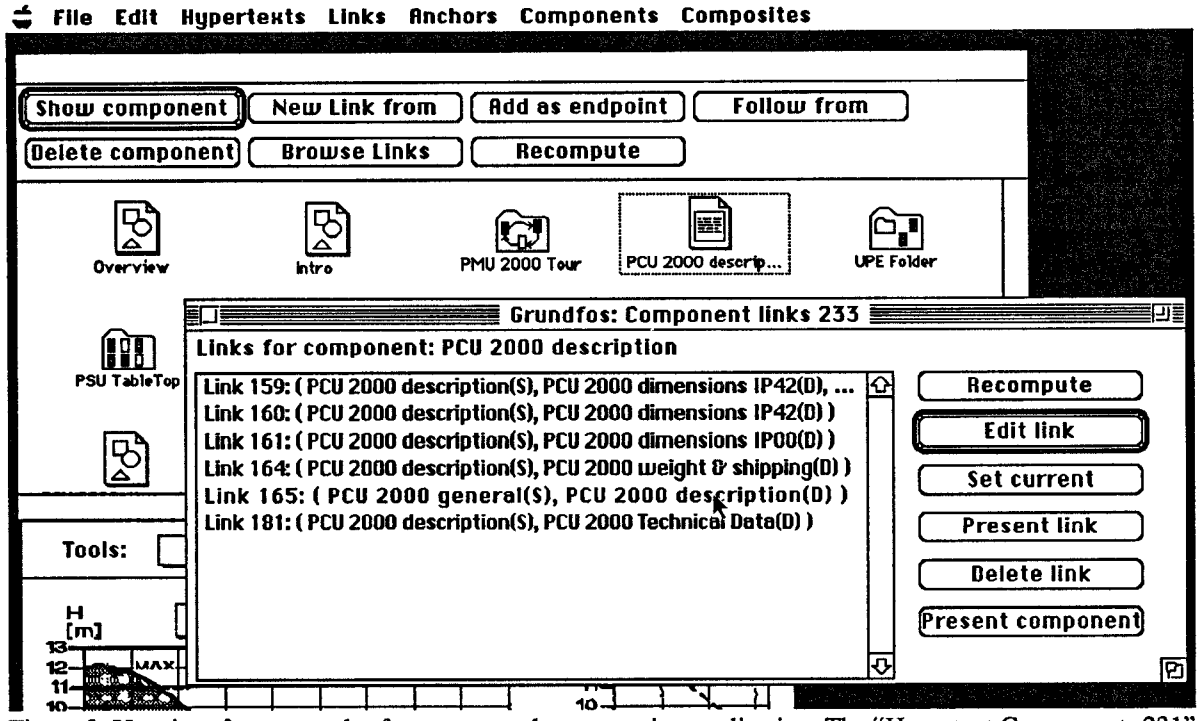

Figure 3: User interface examples from a pump documentation application. The "Hypertext Components 231" window represents a browser of all Atomic and Composite components (using a NonLinkComposite). Folder icons with symbols are used to represent different types of composite components. The "Component links 233" window represents a computed browser (using a LinkComposite) of all the links to and from the text component "PCU 2000 description".

created as a virtual computed composite should become a static structure to be saved persistently, procedures are available for changing the status. Finally, when a link is established to a virtual composite or when a virtual composite is being added to another composite, it is automatically stored by the OODB that traverses transitive closures of object references at save time.

The usage of this generic composite for browsers and queries is discussed by means of some examples. DHM provides a number of different browsers that are distinguished by which filtering of components they apply for their computation. Examples of non-link component browsers[2] and link browsers are given below. It should be noted here that DHM currently does not provide a user interface presentation for generating global graphical maps of components and links[3]. It is, however, possible to get a local list of links for each component in a browser, see Figure 3.

5.1 Non-Link Component Browsers

In the component inheritance hierarchy (STORAGE layer) depicted in Figure 1, CompositeComponent has a branch of subclasses including a 'NonLinkComposite' and an 'AtomComposite'. These are composites that maintain a reference relation (see Section 4) and which are restricted to reference NonLinkComponents and AtomComponents respectively. These composites can as all other composites be assigned a virtual status and be used by several runtime composite instantiations that determine a computation procedure.

In the Instantiation inheritance hierarchy (RUNTIME layer) depicted in Figure 2, there are four subclasses of the NonLinkCompositeInst class: SearchNonLinksInst, SearchAtomsInst, HypertextNonLinksInst, and HypertextAtomsInst that uses the above mentioned composites for computing browsers. This in turn gives four different browsers, see Table 5. Figure 3 shows an example of an icon based presentation for a HypertextNonLinksInst.
The SearchNonLinksInst and the SearchAtomsInst use an attribute on their corresponding composite to store the query for later recomputation.

[2] Traditionally called node browsers.

[3] One of the reasons for not providing global graphical maps is that is a potentially hard problem to design automatic layout algorithms for displaying multi-headed links.

SearchNonLinksInst	Performs a search given some query over the entire set of atomic and composite components
SearchAtomsInst	Performs a similar search, but restricted to the set of atomic components by using the AtomComposite component.
HypertextNonLinksInst	Collects all atomic and composite components in a hypertext
HypertextAtomsInst	Collects only the atomic components in a hypertext by using the AtomComposite component.

Table 5: Examples of browser instantiations.

Subclasses vs. Parameters

In the examples described above a "subclassing" approach has been taken to specify a restriction from non-link components to atomic components. This could also have been done by designing, e.g. the SearchNonLinksInst to take a parameter that tells it to restrict its computation to only collect atomic components and stuff those into the more general NonLinkComposite. This will in most cases be the simplest solution. However, if we want to utilize the knowledge that our composite only contains atomic components, the subclass approach is needed. This is the case if we want to scan the composite contents and perform procedures specific to each atomic component. Our strongly typed language [16] supports this conveniently by making a composite subclass with its contents restricted to atomic components.

The subclassing approach is disadvantageous when users want to take a composite and modify it themselves, e.g. by adding other types of components to it. In such cases parametrized composites are more well suited. The choice between subclassing and use of parameters is open to the user of the DHM framework.

5.2 Link Browsers

The DHM framework also provides a 'LinkComposite' that supports reference relations and is restricted to reference LinkComponents, see Figure 1. This composite subclass is intended to be used for various browsers and queries that collect links from the network.

To handle the behaviour for LinkComposites at runtime, a general LinkCompositeInst and three subclasses are provided. The HypertextLinksInst subclass can be used for a browser that collects all links in a given hypertext. The ComponentLinksInst subclass can be used for a browser that collects all links for a given "root" component, see Figure 3. The root component is represented as an attribute in the LinkComposite, hence the composite can recompute itself again. The AnchorLinksInst subclass can be used for a browser that collects all links for a given anchor in a component. Similarly in this case, the root anchor is represented in an attribute to allow for recomputation. The three LinkCompositeInst subclasses all uses the LinkComposite, i.e. this is an example of different Instantiation types using the same component type. This makes sense because the storage model for all three is the same.

6 COMPOSITES FOR COMMUNICATING HYPERMEDIA STRUCTURES

Previous research in hypermedia has identified a need for communicating trails [4], paths [19], or guided tours [18] between hypermedia users. The analysis work carried out in a branch of our EuroCODE project [1], identified similar needs when designing hypermedia support for a hospital setting. Doctors and radiologists in some situations need means to asynchronously communicate so-called "demonstrations" to each other. A demonstration consists of a sequence of bundles of pictures (X-ray, CT scans, MR scans, etc.), pages from a case record, and Dictaphone messages. This hospital scenario calls for supporting the advanced kind of trails provided by Trigg's [18] GuidedTours and TableTops. A TableTop in this use setting should act as a so-called "snapshot composer": a configuration of presented picture components, text components, and a Dictaphone (sound) component are positioned on the screen, then a snapshot of this configuration is stored in a TableTop. In this scenario, a GuidedTour constructed as a linear (non-branching) ordering of snapshots is sufficient.

Having identified the above requirements for communicating hypermedia structures we designed and implemented a notion of GuidedTours and TableTops in the DHM framework. In contrast to Trigg's NoteCards implementation using links to couple cards to TableTops and TableTops to GuidedTours, specializations of the generic composite classes were used for the implementation.

A TableTopComposite is introduced as a non-virtual composite with no restriction on contents, i.e. it may contain any object being an instance of the component class and its subclasses, see Figure 1.

A TableTopInst class is provided to handle the behaviour of TableTopComposite objects. The 'isComputed' flag is not set for the TableTopInst. The RecomputeComponent virtual is, however, further bound to implement the snapshot mechanism to be invoked "manually" by the user. When invoked, the RecomputeComponent procedure visits all open instantiations and calls AddComponent with the corresponding component reference as a parameter. Moreover, the TableTopInst supports presenting and unpresenting the components referenced by the TableTopComposite in unit operations. An example user interface is shown in Figure 4.

A GuidedTourComposite is implemented as a non-virtual composite with its contents restricted to contain a structured collection of components of type TableTopComposite (see Figure 1). It maintains a reference relation to the TableTopComposites. Its contents consist of a structured

set of TableTopComposites that can be presented and closed one at a time.

A GuidedTourInst class is provided to handle the behaviour of GuidedTourComposites (see Figure 2). It provides support for stepping through an ordered collection of TableTopComposites and presenting them one at a time similar to NoteCards GuidedTour cards. Thus the GuidedTourInst class adds procedures such as 'Start', 'Next, 'Previous', 'Jump', and 'Reset' to the generic CompositeInst class. An example of a preliminary user interface is depicted in Figure 4.

Communicating GuidedTourComposites in DHM

In a shared hypermedia network [5], a GuidedTourComposite can be communicated by telling other users how to find it, e.g. by making a link between it and a common "bulletin board" component or by adding it to a shared composite acting as an "in box" for a receiving user. A GuidedTourComposite and its transitive closure may, however, also be sent to a remote user. This can be done by means of the general partial interchange mechanism for composites, mentioned in Section 3.5.

New Role for Presentation Specifications

When using composites to implement TableTops, it appears that the Dexter notion of presentation specifications ("PSpecs" in short) should also be made applicable to composite "pointers". DHM applies PSpecs to capture presentation information such as window position and size. The Dexter model provides a PSpec for each component, moreover each specifier (link end-point) has a PSpec that can be applied in conjunction with the PSpec of the component referenced by the specifier. Taking window position as an example, a component may store a default window position in its PSpec to be used for presentation, but this position specification can be overruled or combined by the specifier PSpec when the component is presented through following the link. For TableTops a similar mechanism is desirable, since position and size of presentation windows are crucial when preparing a comprehensible TableTop for a GuidedTour. Thus the DHM composite class is extended to optionally hold PSpecs for each component in its contents. This feature is particularly useful for the TableTopComposite subclass, implying that a component being referenced by several TableTops may be presented in windows at a different position for each TableTop.

7 CONCLUSION

This paper discussed different notions of composites in a Dexter-based hypermedia development framework. The original Dexter notion of composite is primarily aimed at handling nodes with structured data objects, and it comes short in handling structures in the hypermedia network itself. Thus a new notion of composite was introduced in the DHM framework. This extended composite concept was presented in terms of generic CompositeComponent and CompositeInst classes. The power of these generic classes

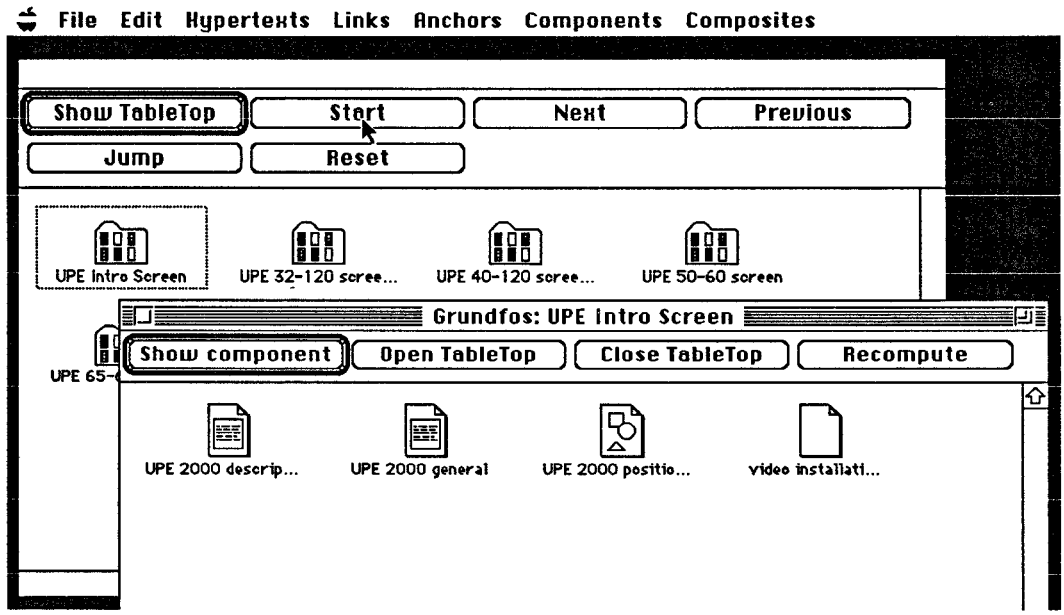

Figure 4: Example from a pump documentation application. The window "UPE 2000 tour" represents a non-branching GuidedTour (using a GuidedTourComposite) of a specific class of pumps. The sequence of the guided tour is currently given by the left-to-right sequence of icons in the tour window – a graph interface is currently under development. The "UPE Intro screen" window represents the first TableTop (a TableTopComposite) in the "UPE 2000 tour".

has been demonstrated by showing how they can easily be specialized to support a rich variety of structuring mechanisms: hierarchy by reference, hierarchy by inclusion, virtual computed composites for browsers and queries, and static composites implementing GuidedTours and TableTops. The composites treated in this paper are only examples, many other hypermedia structures such as bookmarks, hotlists, recent lists, etc. may also be implemented by means of the generic composite classes of DHM.

The status of development of composites in the DHM framework is that the STORAGE and RUNTIME layer classes presented in Figure 1 and 2 are implemented. The presentation user interface is still under development, and it may vary depending on the application domain. Most of the composites discussed in this paper will be further developed and used in our ongoing EuroCODE project.

Topics for further research include graph based browsers, and user interfaces to deal with virtual and computed properties of composites, such that users can distinguish type and status of composites easily.

ACKNOWLEDGEMENTS

Thanks to Randy Trigg for his inspiration and comments on an earlier version of the paper. Thanks to the anonymous reviewers for helpful comments. The work is supported by the Danish Research Programme for Informatics, grant number 5.26.18.19, and the ESPRIT projects EuroCoOp and EuroCODE.

REFERENCES

1. Aas, G., Holmes, P., Lovett, H., Sørgaard, P., Madsen, K.H., and Sandvad, E., *Deliverable D-5.2: Design of the Middle Road Demonstrator*. 1993, Norwegian Computing Center, Oslo, Norway.:
2. Akscyn, R.M., McCracken, D.L., and Yoder, E.A., *KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations*. Communications of the ACM, 1988. 31(7): p. 820-835.
3. Andersen, P., Brandt, S., Hem, J.A., Madsen, O.L., Møller, K.J., and Sloth, L., *Workpackage WP5 Task T5.4, Deliverable D5.4: Distributed Object- Oriented Database Interface*. 1992, Jutland Telephone and Aarhus University:
4. Bush, V., *As We May Think*. The Atlantic Monthly, 1945. August.
5. Grønbæk, K., Hem, J.A., Madsen, O.L., and Sloth, L., *Cooperative Hypermedia Systems: A Dexter-Based Architecture*. Communications of the ACM, 1994. 37(2): p. 64-75.
6. Grønbæk, K. and Malhotra, J. *Building Tailorable Hypermedia Systems: the embedded-interpreter approach*. in *ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '94)*. 1994. Portland, Oregon, US, 23-27 October, 1994: ACM.
7. Grønbæk, K. and Sloth, L., *Hypermedia interchange issues beyond the choice of tagging language*. 1994, Computer Science Department, Aarhus University:
8. Grønbæk, K. and Trigg, R.H., *Design issues for a Dexter-based hypermedia system*. Communications of the ACM, 1994. 37(2): p. 40-49.
9. Halasz, F. and Schwartz, M. *The Dexter Hypertext Reference Model*. in *NIST Hypertext Standardization Workshop*. 1990. Gaithersburg, Md.:
10. Halasz, F. and Schwartz, M., *The Dexter Hypertext Reference Model*. Communications of the ACM, 1994. 37(2): p. 30-39.
11. Halasz, F.G., *Reflections on NoteCards: Seven issues for the next generation of hypermedia systems*. Communications of the ACM, 1988. 31(7): p. 836 -852.
12. Halasz, F.G., Moran, T.P., and Trigg, R.H., *NoteCards in a Nutshell*, in *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*. 1987, p. 45-52.
13. Hardman, L., Bulterman, D.C.A., and van Rossum, G., *The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model*. Communications of the ACM, 1994. 37(2): p. 50-63.
14. Hem, J.A., Madsen, O.L., Møller, K.J., Nørgaard, C., and Sloth, L., *Workpackage WP5 Task T5.2, Deliverable D5.2: Object-Oriented Database Interface*. 1991, Jutland Telephone and Aarhus University,:
15. Knudsen, J.L., Löfgren, M., Madsen, O.L., and Magnusson, B., *Object-Oriented Software Development Environments - The Mjølner Approach*. 1993, Englewood Cliffs, NJ: Prentice Hall.
16. Madsen, O.L., Møller-Pedersen, B., and Nygaard, K., *Object-Oriented Programming in the Beta Programming Language*. 1993, Reading, MA: Addison-Wesley.
17. Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, H., and Thüring, M. *SEPIA a Cooperative Hypermedia Authoring Environment*. in *European Conference on Hypertext (ECHT '92)*. 1992. Milano, Italy: ACM.
18. Trigg, R.H., *Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment*, in *Proceedings of ACM CSCW'88 Conference on Computer-Supported Cooperative Work*. 1988, p. 216-226.
19. Trigg, R.H. and Weiser, M., *TEXTNET: A network-based approach to text handling*. ACM Transactions of Office Information Systems, 1986. 4(1): p. 1-23.
20. Yankelovich, N., Haan, B.J., Meyrowitz, N.K., and Drucker, S.M., *Intermedia: The Concept and Construction of a Seamless Information Environment*. IEEE Computer, 1988. 21(1): p. 81-96.