Hypertext

A Case Study of Formal Object-Oriented Software Development

Andreas Rüping

Forschungszentrum Informatik (FZI)
Bereich Programmstrukturen
Haid-und-Neu-Straße 10-14
D-76131 Karlsruhe

e-mail: rueping@fzi.de

Abstract

Formal object-oriented software development integrates the techniques of object-orientation and formal methods and thus combines the advantages of both. This paper presents a hypertext system as a case study of formal object-oriented software development.

We describe the development of a hypertext system, beginning with informal steps towards an object-oriented model and concluding with the application of formal methods. The case study covers object-oriented analysis and design, formal specification, consistency proof of the specification, implementation, and correctness proof of the implementation.

1 Introduction

Object-orientation provides a large set of concepts and mechanisms for software development, such as classification, object identity, inheritance, and polymorphism. These bring about several methodological advantages. Object-orientation is considered to support in particular comprehensibility, correctness, extensibility, and reusability of software.

However, a sound semantic basis for these mechanisms is required because they can then be used more precisely (cf. [10]). Formal methods cover various techniques that all exploit the advantages of formal semantics. These techniques include formal specification and verification. Formal specification allows for the precise description of a program or a part of it. Verification leads to proofs of certain properties of a program such as consistency or semantically meaningful class relationships.

In this paper we present a hypertext system as a case study of formal object-oriented software development. The example was originally presented in [5]. With this case study we demonstrate how the integration of object-orientation and formal methods combines the advantages of both.

We begin the development of the hypertext system with a requirement analysis, followed by object-oriented modelling and design. Next, we derive a formal specification from the design model and prove the consistency of the specification. Finally, we give an implementation and show how to prove its correctness with respect to the specification.

Among several object-oriented design methods, we choose *Responsibility-Driven Design*. The case study demonstrates how this method works. Furthermore, the case study evaluates how well the design results can be used for setting up a formal specification. Finally, we investigate the specifics of applying *formal* methods to *object-oriented* programming.

The paper is organized as follows. Section 2 deals with object-oriented analysis and design. In Section 3 we discuss how the object-oriented model can be formalized and present a formal specification. We demonstrate how to prove the consistency of a specification and semantically meaningful relationships between classes in Section 4. We conclude with some evaluating remarks on our case study.

2 Analysis and Design

We start with the requirement analysis and modelling of the hypertext system. Throughout these development stages, we apply *Responsibility-Driven Design*, the approach introduced by Rebecca Wirfs-Brock et al. (cf. [1], [15]).

2.1 Requirement Specification

The following is a brief requirement specification of our hypertext system - informal and as general as possible.

> A hypertext system is a collection of information units that are organized not sequentially, but in a graph-like structure. The information units may themselves be structured, that is, nesting and subdividing is generally possible.
>
> No assumptions are made as to what kind of information is handled. For instance, the information units can be filled with texts or images.
>
> The information units are stored in the nodes of such a graph. The links of the (hyper-) graph connect two (or more) information units thus providing the graph-like structure of the information.
>
> Interaction with a hypertext system is possible. Those parts of the system that the user is currently interacting with are called the active parts. The links can be used to travel through the whole structure. Information units can be manipulated by the user who can change their structure and their contents.
>
> A hypertext system can be displayed, at least partially. Therefore, a textual or graphical representation of its contents is required.

Figure 1 gives an sketch of the graph-like structure of a hypertext system. Several nodes are shown that all store some information. Arrows indicate the links between the information units which allow for navigating through the hypertext. Typically, only a few nodes are active at one time. In this example only one node (the one printed black) is active and thus considered to be displayed on the screen.

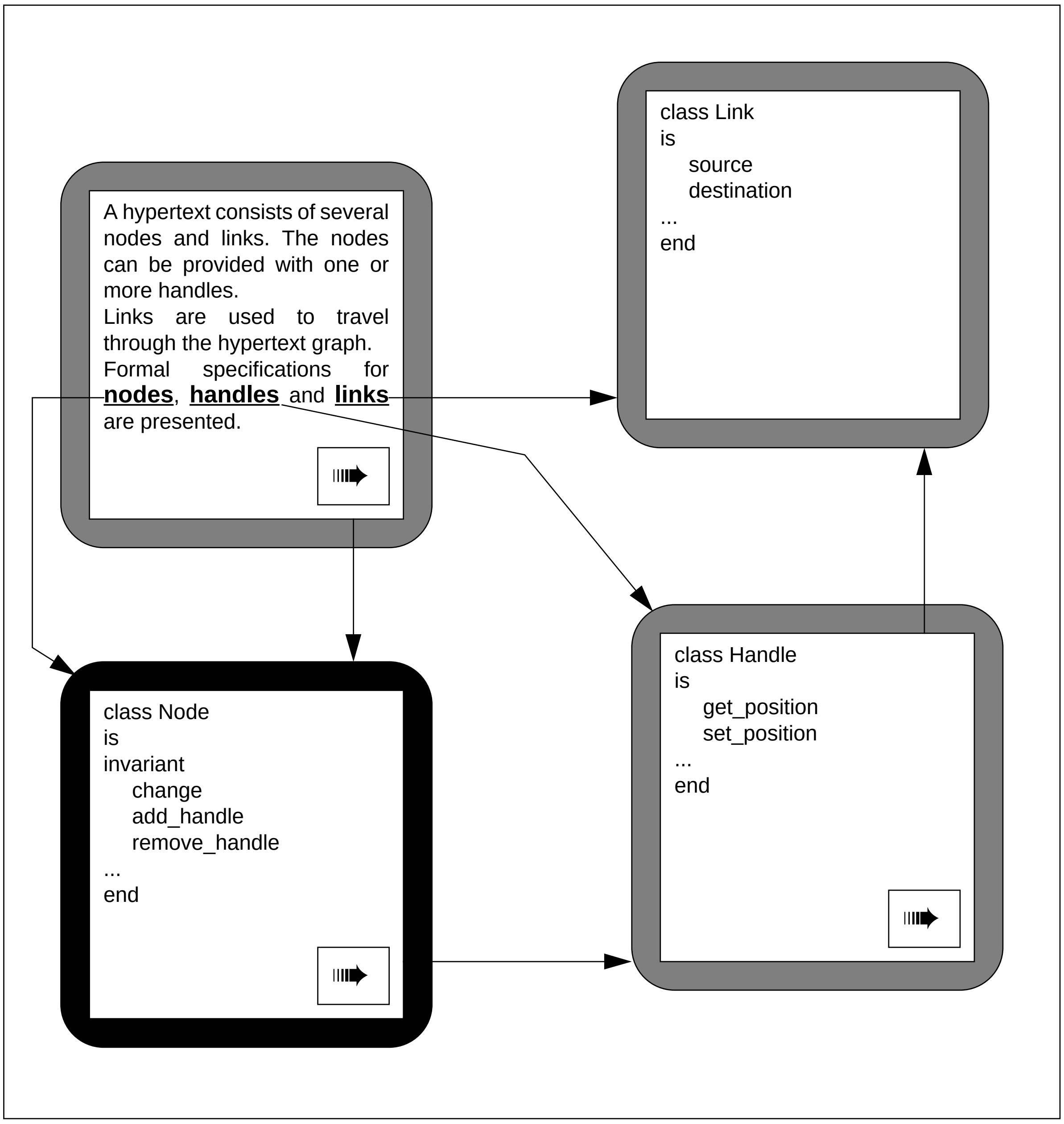
A hypertext consists of several nodes and links. The nodes can be provided with one or more handles.
Links are used to travel through the hypertext graph.
Formal specifications for nodes, handles and links are presented.
class Link
is
source
destination
...
end
class Node
is
invariant
change
add_handle
remove_handle
...
end
class Handle
is
get_position
set_position
...
end

Figure 1 Hypertext

2.2 Modelling the Hypertext System: Classes

Modelling according to Rebecca Wirfs-Brock et al. consists of mainly the following steps:

- finding and recording classes,
- identifying and assigning responsibilities,
- finding and recording collaborations,
- building hierarchies,
- identifying subsystems,

- specification and implementation.

A first list of candidate classes can be derived from the requirement specification. Noun phrases are likely to become items in our model. Thus we obtain the following list from the above requirement specification.

Hypertext System	Text	Hypergraph	Structure
Collection	Image	Interaction	User
Information Unit	Node	Part	Contents
Graph-Like Structure	Graph	System	Representation
Information	Link	Active Part	

Some of these candidates can easily be discarded. We apply the following rules from [15].

- “Do not use more than one word for one concept.”

  We can therefore discard *Information Unit*, *Graph-Like Structure*, and *System*, since *Information*, *Graph*, and *Hypertext System* will do. Furthermore, *Hypertext System* is renamed to simply *Hypertext*.

- “Model the values of attributes of objects, but not the attributes themselves.”

  The contents of a node are some sort of information, hence *Contents* is rather an attribute than a value. *Part* and *Active Part* are neither values, they only refer to a subsystem of a hypertext.

- “Do not model things which are outside the system.”

  The *User* is certainly outside the hypertext system and can be omitted. *Collection*, *Graph*, *Hypergraph*, and *Structure* refer to data structures that we do not model within our system.

We therefore obtain the following tentative list of classes.

Hypertext	Text	Node	Interaction
Information	Image	Link	Representation

We can now begin to set up a class structure. We can group related classes and we can identify subclass / superclass relationships.

As with many interactive software systems, grouping the candidate classes into the three categories *Model*, *View*, *Controller* seems appropriate, similar (although not equal) to a commonly used design pattern (cf. [4]).

- Model: *Hypertext, Node, Link, Information, Text, Image*

  The model includes all classes that provide the real data, in contrast to temporary information or data that is merely essential for the representation of the model.

- View: *Representation*

  All classes that deal with the representation of the model are view classes.

- Controller: *Interaction*

User interaction and operations that invoke changes on the model belong to the controller.

Furthermore, we are able to identify both *Text* and *Image* as possible subclasses of the more general class *Information*.

Rebecca Wirfs-Brock et al. recommend the use of index cards (also called CRC cards) to collect the information we have obtained on the classes of our hypertext system. Each CRC card is meant to supply all information for one class. This includes the class name, the super- and subclasses, and a brief informal description (cf. Figure 2). At later stages of the design we will add further information, such as responsibilities, collaborations, and the like.

2.3 Modelling the Hypertext System: Responsibilities

In [15], Rebecca Wirfs-Brock et al. state:

> Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports. When we assign responsibilities to a class, we are stating that each and every instance of that class will have those responsibilities, whether there is just one instance or many.

Responsibilities include both the knowledge an object maintains and the publicly available actions an object can perform. The requirement specification and the list of classes are sources where we can find possible responsibilities. Verbs in the requirement specification often point to responsibilities. Furthermore, a class once being identified, at least one responsibility for it will typically be thought of when the need for this class was recognized.

In the requirement specification of the hypertext system we find the following “interesting” verb phrases:

- A hypertext system *is* a collection of information units ...
- The information units are *organized* in a graph-like structure.
- The information units can be *filled* with texts or images.
- The information units are *stored* in the nodes of such a graph.
- The links of the (hyper) graph *connect* two (or more) information units ...
- ... parts of the system that the user is currently *interacting* with ...
- The links can be used to *travel* through the whole structure.
- The information units can be *manipulated* by the user
- ... who can *change* the structure and contents of information units.
- A hypertext system can be *displayed*, at least partially.

We now identify several responsibilities and assign them to classes of our model. We intend to distribute these responsibilities over the whole system as evenly as possible. This, however, is hard to achieve and some classes will nevertheless receive more responsibilities than others.

There are several verb phrases in the above list which say that an object is a collection of other objects or has contents. For all these objects knowledge of their contents is required. For instance, a hypertext has to know its nodes and links. A node has to know the information it

Class:	Hypertext
Superclasses:	-
Subclasses:	-
Group:	Model
A hypertext consists of nodes that store information and links that connect information. Parts of a hypertext can be displayed; interaction with a hypertext is possible.	

Class:	Node
Superclasses:	-
Subclasses:	-
Group:	Model
Nodes store the information inside a hypertext.	

Class:	Link
Superclasses:	-
Subclasses:	-
Group:	Model
Links connect information units. Navigation along the links is possible.	

Class:	Information
Superclasses:	-
Subclasses:	Text, Image
Group:	Model
Information is what is stored in (the nodes of) a hypertext, for instance texts or images or something else.	

Class:	Text
Superclasses:	Information
Subclasses:	-
Group:	Model
one particular sort of information ...	

Class:	Image
Superclasses:	Information
Subclasses:	-
Group:	Model
one particular sort of information ...	

Class:	Representation
Superclasses:	-
Subclasses:	-
Group:	View
Parts of hypertext can be displayed. Its contents are then represented textually or graphically on the screen.	

Class:	Interaction
Superclasses:	-
Subclasses:	-
Group:	Controller
To navigate through a hypertext or to change the contents of a hypertext, the user can interact with it.	

Figure 2 CRC cards for the hypertext system

contains and a link has to know its source and destination. Furthermore, not only knowledge of the contents but also the ability to change the contents is necessary.

Such changes are, however, always invoked by an action performed by a user. An interaction object has to pass such a user command to the hypertext model. A user can also navigate through the hypertext. Thus we have to assign to the hypertext representation the responsibility of knowing which parts of the hypertext currently have to be displayed and how to display them.

Since the classes *Text* and *Image* are only exemplary specializations of *Information* but do not provide an essential contribution to the modelling of the hypertext system, we omit these classes from now on.

2.4 Modelling the Hypertext System: Collaborations

A class can fulfil a responsibility either by performing the necessary operations itself or by collaborating with other classes. A collaboration is a request from a client to a server to carry out some task so that the client can fulfil its responsibilities properly.

We can identify collaborations if we look for classes that are not able to fulfil all their responsibilities themselves. For instance, *Interaction*, as a controller class, does not itself perform user commands, but invokes methods of *Hypertext* or *Representation* which may then change contents or activate parts of the hypertext. Thus we have a collaboration with *Interaction* being the client and *Hypertext* and *Representation* being servers.

An updated version of the CRC cards, now including responsibilities and collaborations, is given in Figure 3. The responsibilities of each class are listed on the corresponding CRC card. To record a collaboration, we write the name of the server class on the card of the client class next to the responsibility which makes the collaboration necessary.

2.5 Modelling the Hypertext System: Hierarchies

An evolutionary development style is one characteristic feature of object-oriented analysis and design. The present stage of design represents a good time to analyse the model we have created so far and, if necessary, change parts of it. Changing parts of a model may include adding or removing classes, responsibilities, collaborations, and the like.

The analysis of the preliminary design includes the search for a reasonable inheritance hierarchy of our model. Such an inheritance hierarchy is meant to provide us with a meaningful structure for our model and thus with a more global and precise understanding of our design.

Rebecca Wirfs-Brock et al. (cf. [15]) recommend several tools that are helpful with the analysis of inheritance relationships:

- hierarchy graphs,
- Venn diagrams,
- contracts,
- collaborations graphs.

In our updated model there are no subclass / superclass relationships between the classes we have set up. Therefore we have a plain class structure and a hierarchy graph or a Venn diagram would not make any sense at the moment.

However, the description of our hypertext system is not yet precise enough. This is a point where a revision can start. Among other things, a link is said to connect information units; but it remains unclear what exactly a link is supposed to do.

In the requirement specification we stated that information units can be structured. For instance, if we speak of an information unit, on the one hand we may mean an entire node, on the other hand just a part of the contents of a node. As a consequence, a link can either be fixed to a node, representing all its contents, or to single items within the contents of a node.

Class:	Hypertext
Superclasses:	-
Subclasses:	-
Group:	Model
know its nodes and links	
change its nodes and links	Node, Link
A hypertext consists of nodes that store information and links that connect information. Parts of a hypertext can be displayed; interaction with a hypertext is possible.	

Class:	Node
Superclasses:	-
Subclasses:	-
Group:	Model
know the information it contains	
change the information it contains	Information
Nodes store the information inside a hypertext.	

Class:	Link
Superclasses:	-
Subclasses:	-
Group:	Model
know its source and destination	
change its source and destination	
Links connect information units. Navigation along the links is possible.	

Class:	Information
Superclasses:	-
Subclasses:	Text, Image
Group:	Model
hold a value	
Information is what is stored in (the nodes of) a hypertext, for instance texts or images or something else.	

Class:	Representation
Superclasses:	-
Subclasses:	-
Group:	View
know the active parts of the hypertext	
display the active parts	
Parts of hypertext can be displayed. Its contents are then represented textually or graphically on the screen.	

Class:	Interaction
Superclasses:	-
Subclasses:	-
Group:	Controller
communicate with the user	
pass commands to the model	Hypert., Node
pass display changes to the view	Representation
To navigate through a hypertext or to change the contents of a hypertext, the user can interact with it.	

Figure 3 CRC cards for the hypertext system

This distinction is essential. For example, if some items within a node are removed, we expect a link which was fixed to such an item to be removed as well. A link connected to an entire node should, however, not be automatically removed even if the whole contents of the node are deleted, unless the node is deleted itself.

The result of this analysis is not yet reflected in our model and thus a revision of our model becomes necessary. First, we have to introduce a class that describes the items within the contents of a node to which a link can be fixed. These items are called handles. Second, since a link can connect both nodes and handles, it is reasonable to introduce another additional class: a common superclass of both *Node* and *Handle* which represents an abstraction of these. This class is called *Vertex*; thus, according to our new terminology, a link connects two vertices.

Due to these additional classes, the class structure of our hypertext model is no longer plain. Figure 4 shows the hierarchy graph of our revised hypertext model.

We conclude our design with a precise list of all contracts and a collaboration graph. A contract defines a set of requests that a client can make of a server. The server is guaranteed to respond

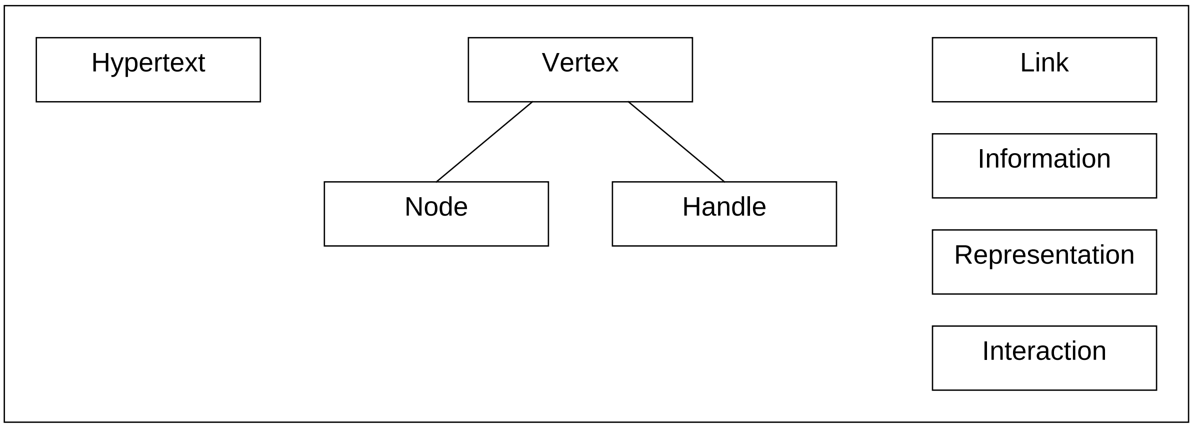

Figure 4 Hierarchy graph of the revised hypertext system

to these requests (cf. [15]). Contracts are usually based on responsibilities that a class does not have for itself but that is needed for a collaboration to be fulfilled.

Figure 5 shows the collaboration graph of our hypertext system. Each box represents a class; nested boxes stand for sub- and superclasses. An arrow between two classes represents a collaboration; the arrow ends in a semicircle which represents a contract. A complete list of all contracts can be found in the final version of our CRC cards presented in Figure 6. All contracts are numbered; those responsibilities that do not have a number are private responsibilities.

This model is, of course, not the most general one. While taking steps towards a higher degree of precision, we have made decisions on the structure of our hypertext system, some of them explicitly and others implicitly.

For instance, we decided that links do connect nodes and handles, but do not connect other links. Thus, hyperlinks are not supported although they were not excluded by the requirement specification. Another example: nodes merely contain information. We do not provide any structuring mechanism for nodes such as, for instance, several information slots within a node.

Decisions like these naturally occur during the design process. It is necessary to regularly check the model obtained so far in order to avoid that the design process takes an unwanted direction. As with other object-oriented design methods, responsibility-driven design is an iterative process that at every stage allows for revising the results of earlier stages. As to our case study, however, we now consider the above model final.

2.6 Subsystems, Protocols, and Implementation

Responsibility-driven design includes further steps towards software development which we omit in our example or deal with in a later section:

- *Subsystems*

  To allow for dealing with large software systems, it is necessary to split up the whole system into subsytems that can be handled more easily. This does not apply to our hypertext system which is too small to be divided into subsystems.

- *Protocols*

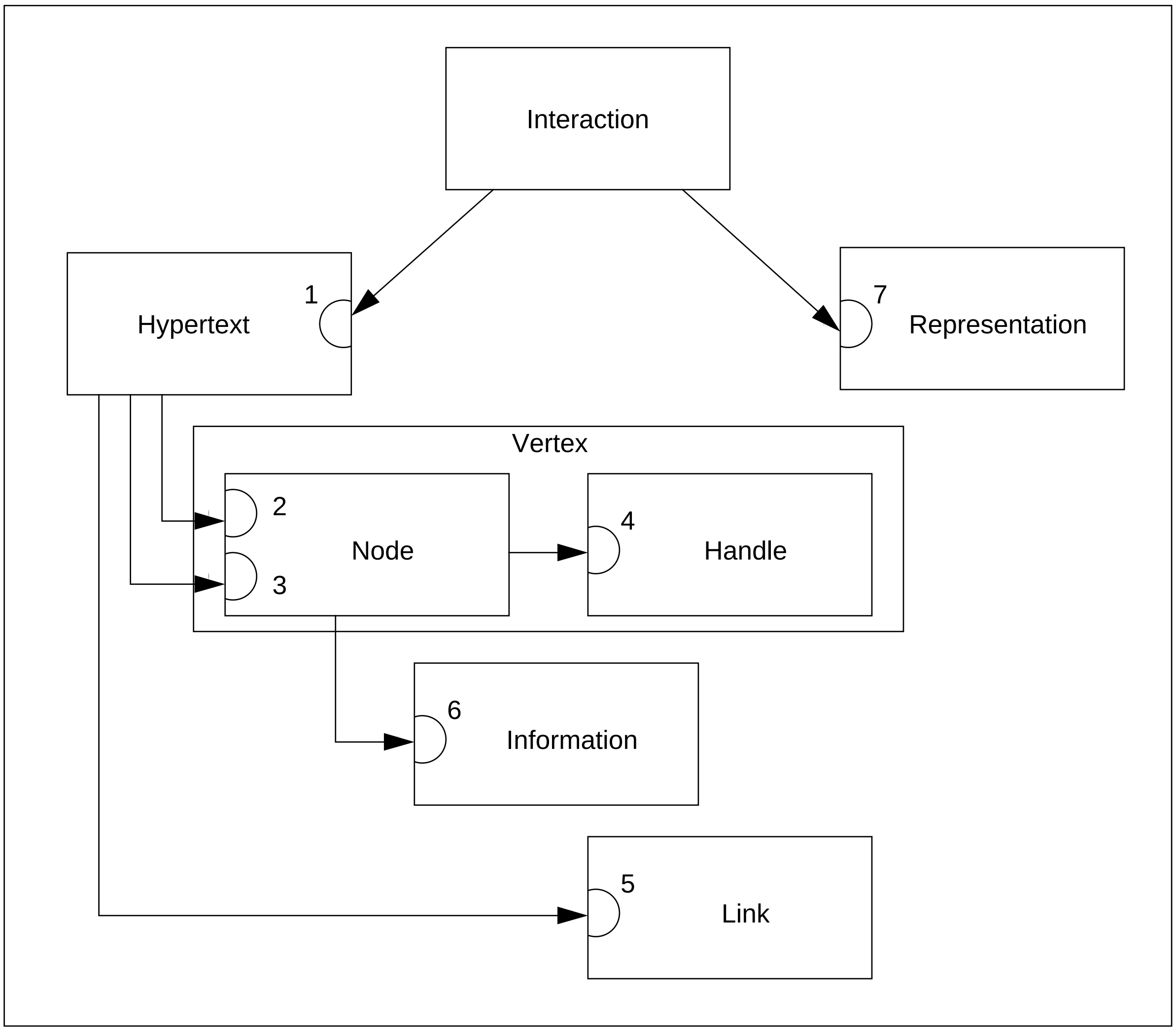

Figure 5 Collaboration graph of the hypertext system

Services supplied by a class are described more precisely. A protocol consists of signatures for all methods of a class. Specifications of both classes and contracts are written.

Since the emphasis of our case study is on formal object-oriented software development, a more formal process is required from this point on. Hence, we do not follow the approach by Rebecca Wirfs-Brock et al. concerning specifications. A formal specification of our hypertext system is presented in the next section.

- *Implementation*

  Software development concludes with an implementation. In [15], several guidelines are presented on how to get an implementation from the design model. In contrast to this, we make an extended use of our formal specification when we give an implementation of the hypertext system. Details are presented later on in this paper.

2.7 Object-Oriented Design: Summary

At the end of the design stage we have obtained a model for our hypertext system upon which we can now build our formal specifications. The model consists of the following parts:

Class:	Hypertext
Superclasses:	-
Subclasses:	-
Group:	Model
know its nodes and links	
1. change its nodes and links	Node (2), (3)
--------------- “ ---------------	Link (5)
A hypertext consists of nodes that store information and links that connect information. Parts of a hypertext can be displayed; interaction with a hypertext is possible.	

Class:	Node
Superclasses:	Vertex
Subclasses:	-
Group:	Model
know its information and handles	
2. change the information	Information (6)
3. change the handles	Handle (4)
Nodes store the information inside a hypertext.	

Class:	Handle
Superclasses:	Vertex
Subclasses:	-
Group:	Model
4. point to information within a node	
Handles describe particular information units within a node.	

Class:	Vertex
Superclasses:	-
Subclasses:	Node, Handle
Group:	Model
serve as source or destination	
Vertex abstracts from Node and Handle. Both the source and the destination of a link are vertices.	

Class:	Link
Superclasses:	-
Subclasses:	-
Group:	Model
know its source and destination	
5. change its source and destination	
Links connect vertices. Navigation along the links is possible.	

Class:	Information
Superclasses:	-
Subclasses:	Text, Image
Group:	Model
6. hold a value	
Information is what is stored in (the nodes of) a hypertext, for instance texts or images or something else.	

Class:	Representation
Superclasses:	-
Subclasses:	-
Group:	View
know active parts of the hypertext	
7. display the active parts	
Parts of hypertext can be displayed. Its contents are then represented textually or graphically on the screen.	

Class:	Interaction
Superclasses:	-
Subclasses:	-
Group:	Controller
communicate with the user	
pass commands to the hypertext	Hypertext (1)
pass user commands to a node	Node (2), (3)
pass display changes to the view	Represent. (7)
To navigate through a hypertext or to change the contents of a hypertext, the user can interact with it.	

Figure 6 CRC cards for the hypertext system

- a list of classes, including an informal description of their purposes,
- a list of responsibilities for each class,
- a list of collaborations with contracts specifying which services each class offers,
- a class hierarchy, representing both sub- / superclass relationships and collaborations.

3 Formal Specification

3.1 From an Informal to a Formal Description

We continue our case study with a formal specification of the hypertext system. Later on, we will give both a consistency proof of the specification and a correctness proof of an implementation with respect to the specification. To keep the case study brief, we now concentrate on the classes of the group *Model* and do not include *Representation* and *Interaction* into the formal specification. Furthermore, we assume an appropriate concrete class to be available to be substituted for *Information*, and omit the formal specification of the latter, too.

The formal specification is based on the design model that we have obtained in the previous section. We are able to derive several characteristics of the specification from this model. Doing this, we apply the following principles:

- *Groups*

  Each group of the design model is represented by a specification module.
- *Classes*

  For each class of the design model a specification class is introduced.
- *Private responsibilities*

  The private responsibilities of a class are represented by attributes or private methods of the corresponding specification class.
- *Contracts*

  Contracts are specified as public methods that can be invoked from a client class and result in the server class discharging its responsibilities with regards to the client.
- *Hierarchies*

  The subclass / superclass relationships within the design model are adopted in the specification and are expressed by inheritance.

Thus we have to introduce the classes *Hypertext*, *Vertex*, *Node*, *Handle*, and *Link* into the specification. Since we concentrate on the group *Model*, our specification will only consist of one module called *Hypertext_Model* to which all mentioned classes belong.

3.2 Specification Techniques

The specification we give is written in ‘Coffer’ (cf. [9]), a state-based, object-oriented language which is built upon the following specification techniques.

- *Modules, Classes, and Instantiation*

A specification in ‘Coffer’ consists of several modules which all contain one or more classes. Objects are global within a module; they are, however, not global within a whole system. This restriction is necessary for both modular consistency proofs and modular correctness proofs which we will discuss in more detail later on in this paper.

Different kinds of classes are distinguished. The main distinction is between abstract classes for the specification and concrete classes for the implementation. For a more detailed discussion of different kinds of classes see [8] and [14].

Abstract classes specify behaviour, mainly by specification statements. No objects can be instantiated from an abstract class; however, the behaviour described by an abstract class can be inherited by other, possibly concrete, classes.

Concrete classes implement the behaviour and the state space of their objects. Objects can be created as instances of a concrete class.

- *Inheritance and Conformance*

  Inheritance is understood as a syntactic notion to express that one class adopts all features (attributes, methods, and invariants) from another class.

  Conformance is a semantic relationship between classes which is usually to be achieved by inheritance. A class conforms to its superclass if an object of the superclass can always be replaced by an object of the subclass. For a more precise definition of subtyping and other semantic class relationships see [12] and [14].

- *Attributes, Methods and Preconditions*

  For each class the necessary attributes and methods are declared. A method declaration consists of a header and a body. The header defines the signature of a method and possibly a precondition (cf. [6]). The body contains specification statements.

- *Specification statements*

  A specification statement is a statement that is not required to be executable. Specification statements can be used to describe the desired result of a statement, for instance by means of a logical predicate. Specification statements can be non-deterministic.

- *Invariants*

  An invariant of a class is a logical predicate that is required to hold before and after the execution of an arbitrary method.

3.3 Object Identities

The objects in our design model are implicitly expected to have identities. This includes pointer semantics for objects. In the formal model we use, however, object identities have to be made explicit because otherwise formal reasoning about the specification would be impossible.

A set of objects which more precisely is indeed a set of pointers to several objects is specified as a mapping from a set of identifiers to a set of object values. This form of specification allows for a formal semantics of statements that include pointers, and thus enables us to reason formally about the behaviour of methods that make use of object identities.

As mentioned above, object identities are global within a module. Therefore, we need global mappings for those classes of a module that have to be provided with object identities (at least for those classes that are indeed instantiated). These mappings map identifiers to object values.

They do so for all objects that have been created so far. For each such class, the name of the corresponding mapping is the name of the class preceded by a '#'. To denote an object value, we write the identifier followed by a '↑'.

Furthermore, each object is implicitly supplied with a read-only attribute *self* that contains its own identifier. Thus, an object knows by which identifier it can be referenced.

In Section 4.4 we will discuss the question why objects identities are global only within one module, but cannot be global within the whole system.

3.4 Formal Specification of the Hypertext System

To reduce the complexity of the specification, we do not introduce object identities for all classes, but only for *Vertex*, *Node* and *Handle*. We do not introduce object identities for *Hypertext* since we only deal with one hypertext object here. The objects of *Link* are considered to be mere values. We assume that there are classes *VertexId*, *NodeId*, and *HandleId* providing identifiers for objects.

The module definition for *Hypertext_Model* is given in Figure 7. From outside the module, only hypertexts can be created; the creation of vertices, nodes, handles, and links is restricted to within the module, as the creation export clause says. The formal specifications of the classes *Hypertext*, *Vertex, Node, Handle*, and *Link* are presented in Figures 8, 9, 10, 11, and 12.

The classes are explained as follows:

- *Hypertext*

  The specification of *Hypertext* is easily derived from the CRC card. A hypertext is required to know about its nodes and links and to be able to change them. The former is carried out by the attributes *nodes* and *links* and the methods *links_from* and *links_to*. For the latter the methods *add_node*, *remove_node*, *add_link*, and *remove_link* are introduced.

  The invariant says that the environments of both the source and the destination of an arbitrary link must be nodes of the hypertext. The precondition of the method *add_link* says that a new link can only be inserted if it respects the invariant.

- *Vertex*

  Vertex is the common superclass of both *Node* and *Handle*. It supplies a method *env* that always returns the environment of the vertex, which is a node identifier.

- *Node*

  From the CRC card we know that a node has some contents and that it must be able to know and change its handles. For the former, the attribute *contents* and the method *change* are introduced, for the latter there are the methods *add_handle* and *remove_handle*. The inherited attribute *environment* holds *self*, the identifier of the node itself.

- *Handle*

  A handle has a position which can be accessed via *get_pos* and *set_pos*. The inherited attribute *environment* points to the node in which the handle is contained; thus a node is the environment of all its handles. (This is expressed by the second part of the invariant for *Node*.)

- *Link*

A link is a pair of two *VertexIds*, called *source* and *destination*, which can be directly accessed. A link can be considered a traditional record consisting of two components.

Creation of objects with identities is done by the method *new* which returns a new identifier for the type given to *new* as a parameter. Also, the pair consisting of the new identifier and the new object is inserted into the appropriate module-global mapping. Each time an object is created, the method *init* of the new object's class is invoked. If *new* has an additional parameter, this additional parameter is passed to *init*.

For instance, the method *add_handle* in *Node* creates a new handle and returns a new identifier for it. The pair consisting of the new identifier and the new object is inserted into *#Handle*. The method *init* is invoked for the new handle and the additional parameter *self* is passed to *init*.

Since we do not create objects of type *Vertex*, there is no mapping *#Vertex* declared for the module *Hypertext_Model*. Furthermore, no initialization method needs to be provided for *Vertex*. We only declare an object of type *Vertex* in cases where we expect either a node or a handle.

Therefore, the classes *Node* and *Handle* do not only inherit from *Vertex*. In the specification, they are also said to conform to *Vertex*. Since conformance cannot be simply declared between two classes but needs to proven, we have a proof obligation here. We will discuss this sort of proof obligation in Section 4.5 of this paper.

```
module Hypertext_Model

include Hypertext, Vertex, Node, Handle, Link
creation export Hypertext

                #Node: Mapping [NodeId, Node]
                #Handle: Mapping [HandleId, Handle]

end Hypertext_Model
```

Figure 7   Module 'Hypertext_Model'

```
class Vertex

     private    environment: NodeId

                env (): nid: NodeId is nid := environment

end Vertex
```

Figure 9   Class 'Vertex'

```
class Link

     source, destination: VertexId

end Link
```

Figure 12   Class 'Link'

```
class Hypertext

        private    nodes: Set [NodeId]
        private    links: Set [Link]

invariant   ∀ lnk ∈ links • con (lnk)

        private    con (lnk: Link) Boolean is
                   lnk.source↑.env ∈ nodes ∧ lnk.destination↑.env ∈ nodes

                   add_node () new-nid: NodeId is
                   new-nid := new (Node); nodes := nodes ∪ {new-nid}

                   remove_node (old-nid: NodeId) is
                   nodes := nodes - {old-nid}; links := links - links_from (old-nid) - links_to (old-nid)

                   add_link (new-lnk: Link) pre con (new-lnk) is
                   links := links ∪ {new-lnk}

                   remove_link (old-lnk: Link) is
                   links := links - {old-lnk}

                   links_from (nid: NodeId) from: Set (Link) is
                   from := {lnk ∈ links | lnk.source↑.env = nid}

                   links_to (nid: NodeId) to: Set (Link) is
                   to := {lnk ∈ links | lnk.destination↑.env = nid}

                   init () is
                   (nodes, links) := ({ }, { })

end Hypertext
```

Figure 8 Class 'Hypertext'

```
class Node

inherit Vertex, conform Vertex

        private    contents: Information
        private    handles: Set [HandleId]

invariant   environment = self ∧
            ∀ hid ∈ handles • hid↑.environment = self

                   change (info: Information) is contents := info

                   add_handle () new-hid: HandleId is
                   new-hid := new (Handle, self); handles := handles ∪ {new-hid}

                   remove_handle (old-hid: HandleId) is handles := handles - {old-hid}

                   init () is (environment, contents, handles) := (self, { }, { })

end Node
```

Figure 10 Class 'Node'

4 Verification

```
class Handle
inherit Vertex, conform Vertex
        private   position: Position
                  get_pos () pos is pos := position
                  set_pos (pos: Position) is position := pos
                  init (created_by: NodeId) is (environment, position) := (created_by, 0)
end Handle
```

Figure 11 Class 'Handle'

4.1 Verification Techniques

This section deals with the verification of our hypertext system. We demonstrate two different proof techniques when proving the following properties:

- *Consistency*

  Consistency of a specification means that the invariants hold after initialization and before and after the execution of a method. To prove consistency, we apply a technique based on weakest preconditions as described in [2], [3], [7], and [11].

  For our specification, consistency proofs can only be given module-wise, not class-wise. This is due to the fact that we have objects that are global within a module. Invariants can rely on such global objects. Thus, the invariant of a class might be violated by the methods of another class unless the opposite is proven. (All invariants do indeed belong to the model and not to a class. They are assigned to classes because there they can be expressed best.)

  Our task is therefore to check for all methods of all classes within a module whether or not they respect all invariants.

- *Conformance*

  Conformance is a semantic relationship between classes. It occurs in two different situations. First, conformance is the intended relationship if a class inherits from another. Second, conformance is required between a specification class and its implementation.

  We make again use of a verification technique based on weakest preconditions (cf. [2], [3], [7], [11]). However, the scope of this paper allows us only to outline conformance proofs.

4.2 Techniques for Consistency Proofs

The weakest precondition wp (P, c) of a method P and a postcondition c is intuitively explained as the weakest condition that we have to assume before the execution of P so that after the execution of P the postcondition c holds.

To prove the consistency of a module, we have to verify the following two logical expressions for each invariant *Inv*:

for each initialization method *init* that may be called from outside the module:

wp (init, Inv) = true

"The invariant is required to hold after the execution of *init*."

and

for each non-initialization method *m*:

Inv $\Rightarrow$ wp (m, Inv)

"Provided the invariant holds before the execution of *m*, it also holds after the execution of *m*."

We have not yet given a formal semantics for the specification in Section 3, but only explained the specification informally. In Figure 13 we now present a formal semantics for the statements used in the specification (and some other), namely in terms of weakest preconditions.

(1)	let p (x) = P:	wp (p (x), c) = $\forall$ x $\bullet$ wp (P, c)
(2)	let p (x) = pre b P:	wp (p (x), c) = $\forall$ x $\bullet$ b $\Rightarrow$ wp (P, c)
(3)	wp (skip, c) = c	
(4)	wp (abort, c) = false	
(5)	wp (magic, c) = true	
(6)	wp (x := e, c) = c [x:=e]	
(7)	wp (P; Q, c) = wp (P, wp (Q, c))	
(8)	wp (if b then P else Q end, c) = (b $\Rightarrow$ wp (P, c)) $\wedge$ ($\neg$b $\Rightarrow$ wp (Q, c))	
(9)	let #T: Mapping [TId, T], let class T have init ():	
	wp (id := new (T), c) = wp (id :- id $\notin$ domain (#T); #T := #T $\oplus$ (id, ?); id$\uparrow$.init (), c) = $\forall$ id $\bullet$ id $\notin$ domain (#T) $\Rightarrow$ wp (#T := #T $\oplus$ (id, ?); id$\uparrow$.init (), c)	
(10)	let #T: Mapping [TId, T], let class T have init (u: U):	
	wp (id := new (T, u), c) = wp (id :- id $\notin$ domain (#T); #T := #T $\oplus$ (id, ?); id$\uparrow$.init (u), c) = $\forall$ id $\bullet$ id $\notin$ domain (#T) $\Rightarrow$ wp (#T := #T $\oplus$ (id, ?); id$\uparrow$.init (u), c)	

Figure 13 Rules for the weakest precondition

- Rules (1) and (2) say that the weakest precondition of a parametrized method *p (x)* and a postcondition *c* has to hold for all valid parameters *x*, possibly restricted to a precondition *b* which has to hold if *p (x)* is invoked.

- Rules (3), (4), and (5) give the weakest preconditions for three very simple statements: *skip* is the empty statement which causes no changes whatsoever, *abort* leads to a totally unpredictable state, and *magic* fulfils all conditions it is expected to fulfil. (Obviously, *magic* cannot be implemented.)

- Rule (6) says that the weakest precondition of an assignment $x := e$ and a postcondition c is c with all occurrences of x replaced by the expression e. Rules (7) and (8) give the weakest preconditions for sequential composition and alternative.
- Rules (9) and (10) define the *new* statement. These rules first give substitutions for wp-expressions, and then compute the weakest precondition. In detail, rule (9) says the following: Declaring a new object as an instance of class *T* means that a yet unused identifier *id* for class *T* is chosen non-deterministicly, the mapping *#T* is extended with an additional pair consisting of *id* and the new object, and the new object is initialized. Rule (10) says the same, but for an initialization method *init* that is parametrized.

It is clear now, why only for initialization methods that can be invoked from outside the module we have to prove that after the execution of an initialization method all invariants hold. Since instances of classes that do not export creation can only be created by methods inside the module, the proof that all other initialization methods respect the invariants is covered by the rules (9) and (10).

4.3 Consistency Proof for the Hypertext System

For our hypertext system, we have to prove that for the method *init* from the class *Hypertext* and all invariants *Inv* of the module *wp (init, Inv) = true* holds. The proof for the invariant assigned to the class Hypertext is given in Figure 14. The proofs for both other invariants are omitted here. They are equally simple because the attributes *nodes* and *links* are initialized to empty sets while the invariants scan all existing nodes, links, and handles via an all-quantification.

In all proofs, transformations are annotated with the number of the rule (cf. Figure 13) that is being applied. Transformations that are not annotated rely on the usual inference rules for logic.

	wp (init, Inv)
=	wp ((nodes, links) := ({ }, { }), Inv)
= (6)	$\forall$ lnk $\in$ { } • lnk.source↑.env $\in$ { } $\wedge$ lnk.destination↑.env $\in$ { }
=	true ■

Figure 14 Consistency proof for ‘init’ (class ‘Hypertext’)

Next we demonstrate several proofs of (non-initialization) methods preserving the invariants. We choose the invariant assigned to the class *Hypertext*:

$\forall$ lnk $\in$ links • lnk.source↑.env $\in$ nodes $\wedge$ lnk.destination↑.env $\in$ nodes

We concentrate on the most interesting proofs and omit ‘trivial’ proofs in those cases where a method has no influence on the objects covered by the invariant. This sort of proof can be easily done applying rule (6) from Figure 13.

We present the following proofs:

- The consistency proof for *links_from* (class *Hypertext*) is given in Figure 15. It is an example of a ‘trivial’ proof that is lead by only applying rule (6) once.
- The consistency proof for *links_to* (class *Hypertext*) can be done analogously and is omitted.

- The consistency proofs for *add_link* and *remove_link* (class *Hypertext*) are given in Figures 16 and 17.
- The consistency proofs for *add_node* and *remove_node* (class *Hypertext*) are given in Figures 19 and 18.
- The consistency proofs for *env* (class *Vertex*), *change*, *add_handle*, *remove_handle* (class *Node*), *get_pos*, and *set_pos* (class *Handle*) are 'trivial' and are therefore omitted.

	wp (from := links_from (nid), Inv)
=	wp (from := $\{\text{lnk} \in \text{links} \mid \text{lnk.source}\uparrow\text{.env} = \text{nid}\}$, Inv)
= (6)	Inv ■

Figure 15 Consistency proof for 'links_from'

	wp (add_link (new-lnk) pre con (new-lnk), Inv)
= (2)	$\forall$ new-lnk $\bullet$ (con (new-lnk) $\Rightarrow$ wp (links := links $\cup$ {new-lnk}, Inv))
= (6)	$\forall$ new-lnk $\bullet$ (con (new-lnk) $\Rightarrow$ $\forall$ lnk $\in$ links $\cup$ {new-lnk} $\bullet$ con (lnk)
=	$\forall$ new-lnk $\bullet$ (con (new-lnk) $\Rightarrow$ $\forall$ lnk $\in$ links $\bullet$ con (lnk)
$\Leftarrow$	$\forall$ new-lnk $\bullet$ $\forall$ lnk $\in$ links $\bullet$ con (lnk)
=	$\forall$ lnk $\in$ links $\bullet$ con (lnk)
=	Inv ■

Figure 16 Consistency proof for 'add_link'

	wp (remove_link (old-lnk), Inv)
=	wp (links := links - {old-lnk}, Inv)
= (6)	$\forall$ lnk $\in$ links - {old-lnk} $\bullet$ con (lnk)
$\Leftarrow$	$\forall$ lnk $\in$ links $\bullet$ con (lnk)
=	Inv ■

Figure 17 Consistency proof for 'remove_link'

There are two more invariants specified in the module *Hypertext_Model* (for which we do not present the consistency proofs in this paper):

environment = self (in class *Node*)

and

$\forall$ hid $\in$ handles $\bullet$ hid$\uparrow$.environment = self (in class *Node*)

The consistency of all methods with respect to these two invariants can be verified in a similar way as with the other invariant. The proof techniques are essentially the same. Most of the proofs are trivial; some, however, are not. For instance, to verify that *add_handle* respects the second of the above invariants requires a proof as difficult as the one in Figure 19. In both cases, the

$$
\begin{array}{ll}
 & \text{wp (remove\_node (old-nid), Inv)} \\
= & \text{wp (nodes := nodes - \{old-nid\};} \\
 & \quad \text{links := links - links\_from (old-nid) - links\_to (old-nid), Inv)} \\
= (7) & \text{wp (nodes := nodes - \{old-nid\},} \\
 & \quad \text{wp (links := links - links\_from (old-nid) - links\_to (old-nid), Inv))} \\
= (6) & \text{wp (nodes := nodes - \{old-nid\},} \\
 & \quad \forall\ \text{lnk} \in \text{links - links\_from (old-nid) - links\_to (old-nid)} \bullet \\
 & \qquad \text{lnk.source}\uparrow\text{.env} \in \text{dom (nodes)} \wedge \text{lnk.destination}\uparrow\text{.env} \in \text{dom (nodes))} \\
= (6) & \forall\ \text{lnk} \in \text{links - links\_from (old-nid) - links\_to (old-nid)} \bullet \\
 & \quad \text{lnk.source}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \wedge \\
 & \quad \text{lnk.destination}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \\
= & \forall\ \text{lnk} \in \text{links - \{lnk} \in \text{links | lnk.source}\uparrow\text{.env = old-nid} \vee \text{lnk.destination}\uparrow\text{.env = old-nid\}} \bullet \\
 & \quad \text{lnk.source}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \wedge \\
 & \quad \text{lnk.destination}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \\
= & \forall\ \text{lnk} \in \text{links} \bullet \text{(lnk.source}\uparrow\text{.env} \neq \text{old-nid} \wedge \text{lnk.destination}\uparrow\text{.env} \neq \text{old-nid)} \Rightarrow \\
 & \quad \text{lnk.source}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \wedge \\
 & \quad \text{lnk.destination}\uparrow\text{.env} \in \text{nodes - \{old-nid\}} \\
= & \forall\ \text{lnk} \in \text{links} \bullet \text{(lnk.source}\uparrow\text{.env} \neq \text{old-nid} \wedge \text{lnk.destination}\uparrow\text{.env} \neq \text{old-nid)} \Rightarrow \\
 & \quad \text{lnk.source}\uparrow\text{.env} \in \text{nodes} \wedge \\
 & \quad \text{lnk.destination}\uparrow\text{.env} \in \text{nodes} \\
\Leftarrow & \forall\ \text{lnk} \in \text{links} \bullet \\
 & \quad \text{lnk.source}\uparrow\text{.env} \in \text{nodes} \wedge \\
 & \quad \text{lnk.destination}\uparrow\text{.env} \in \text{nodes} \\
= & \text{Inv} \ \blacksquare
\end{array}
$$

Figure 18 Consistency proof for ‘remove_node’

complexity of the proofs is caused by the rather difficult way in which the *new* statement must be handled formally.

4.4 Object Identities (Revisited)

It now becomes obvious why we had to restrict object identities to be global only within a module. If we had allowed for objects to be referenced by global identifiers from an arbitrary class within the whole system, we would have had to prove that all methods in all classes preserve all invariants. Thus, modular verification would have become impossible.

The central point of fixing object identities to modules is that both object identifiers and object values are always kept in the same module. Objects can be referenced via their identifiers only in the same module that stores the object values.

As a consequence, cyclic pointer structures are only possible within a module. For instance, a node contains several handles. Each handle possesses a reference to the node in which it is contained, namely by the attribute *environment*. This is specified by a cyclic pointer structure which cannot be distributed over several modules.

However, it is of course possible to create objects of a class outside the module of this class, as long as both the object identifiers and the object values are defined in the same external module. For example, the hypertext controller will probably store one or more hypertexts (instances of the class *Hypertext*) and refer to them via identifiers. This is legal since the class *Hypertext* is listed in the creation export clause of *Hypertext_Model*.

```
             wp (new-nid := add_node (), Inv)
=            wp (new-nid := new (Node); nodes := nodes ∪ {new-nid}, Inv)
= (7)        wp (new-nid := new (Node), wp (nodes := nodes ∪ {new-nid}, Inv))
= (6)        wp (new-nid := new (Node),
                       ∀ lnk ∈ links •
                             lnk.source↑.env ∈ nodes ∪ {new-nid} ∧
                             lnk.destination↑.env ∈ nodes ∪ {new-nid})
= (9)        ∀ new-nid • new-nid ∉ domain (#Node) ⇒
                  wp (#Node := #Node ⊕ (new-nid, ?); new-nid↑.init (),
                       ∀ lnk ∈ links •
                             lnk.source↑.env ∈ nodes ∪ {new-nid} ∧
                             lnk.destination↑.env ∈ nodes ∪ {new-nid})
=            ∀ new-nid • new-nid ∉ domain (#Node) ⇒
                  wp (#Node := #Node ⊕ (new-nid, ?);
                       new-nid↑.environment := new-nid↑.self;
                       new-nid↑.contents := { };
                       new-nid↑.handles := { },
                       ∀ lnk ∈ links •
                             lnk.source↑.env ∈ nodes ∪ {new-nid} ∧
                             lnk.destination↑.env ∈ nodes ∪ {new-nid})
= (6), (7)   ∀ new-nid • new-nid ∉ domain (#Node) ⇒
                       ∀ lnk ∈ links •
                             lnk.source↑.env ∈ nodes ∪ {new-nid} ∧
                             lnk.destination↑.env ∈ nodes ∪ { new-nid}
⇐            ∀ new-nid • ∀ lnk ∈ links •
                             lnk.source↑.env ∈ nodes ∪ {new-nid} ∧
                             lnk.destination↑.env ∈ nodes ∪ { new-nid}
⇐            ∀ new-nid • ∀ lnk ∈ links •
                       lnk.source.env ∈ nodes ∧
                       lnk.destination.env ∈ nodes
=            ∀ lnk ∈ links •
                       lnk.source.env ∈ dom (nodes) ∧
                       lnk.destination.env ∈ dom (nodes)
=            Inv ■
```

Figure 19 Consistency proof for ‘add_node’

Summarizing, the concept of modules serving as the scope for object identities brings about the following advantages:

- We are able to use object identities in our specification which is important since object identities are an essential part of the object-oriented paradigm. Object identities are necessary to derive a specification from an object-oriented design model.
- We can use object identities within a module to express (even cyclic) pointer structures.
- Making object identities explicit happens at a central place within a module. Thus, the classes themselves can use object identities without a large notational burden.

- Modular verification is possible. Although verification cannot be done class-wise, it can indeed be done module-wise. This is an important advantage for incremental software development, for instance if program parts (modules in our case) have to verified before they are added to a software library.

4.5 Conformance Proofs (Outline)

As mentioned before, the intuitive understanding of conformance is that if one class conforms to another, objects of the former can in an arbitrary situation be substituted for objects of the latter. Thus, to each method of the more abstract class there must be a corresponding and possibly more deterministic method in the conforming class. Also, the conforming class may have additional methods.

There are two different situations in which a class conforms to another. First, conformance can be declared (and then needs to be proven) between two classes of the specification, in particular, if one class inherits from another. For instance, both *Node* and *Handle* conform to *Vertex*. Second, an implementation is required to conform to its specification.

Although these two situations are totally different from the modelling point of view, the underlying logic is the same in both cases. Thus, the same verification techniques are applied to prove conformance in both situations.

The proof that a (sub-) class *S* with state space X conforms to a class *C* with state space Y requires the verification of the following:

> There is a mapping f: (S, X) $\rightarrow$ (T, Y) mapping the methods of class S to methods of class T and state space X to state space Y, so that the following two conditions hold:
>
> > $\mathrm{Inv}_S (x) \Rightarrow \mathrm{Inv}_T (f (x))$
> >
> > “The invariant of *S* is not weaker than the invariant of *C*.”
>
> and
>
> > for all postconditions *C*, an arbitrary method m_S in *S*, the corresponding method m_T in *T*, and all states *x*, *y*:
> >
> > $\mathrm{Inv}_S (x) \Rightarrow \big(y = f (x) \wedge \mathrm{wp} (m_T, C (y)) \Rightarrow \mathrm{wp} (m_S, C (f (x))) \big)$
> >
> > “Restricted to all states in which the invariant for class *S* holds, the weakest precondition for a method m_S in *T* and an arbitrary postcondition implies the weakest precondition for the corresponding method m_T and the corresponding postcondition in *S*.”

The meaning of the second condition can be explained with the commuting diagram presented in Figure 20. We assume that there is an object *x* of class *S* (which respects the invariant of *S*). Via mapping *f*, we obtain the corresponding object *y* of class *T*. Applying m_T to *y* yields a certain result described by *C (y)*, as long as *wp* (m_T, *C (y))* holds. Conformance says that in this case *wp* (m_S, *C (f (x)))* also holds which means that applying m_S to *x* does indeed lead to a corresponding result *C (f (x))*.

In the specification of our hypertext system, *Node* and *Handle* conform to *Vertex*. Although we do not give a formal proof, we can explain why conformance holds using the above criteria.

In our case, the mapping *f* from *Node* to *Vertex* is very simple. Since no redefinitions are made in *Node*, the mapping *f* maps the inherited method *env* and the inherited attribute *environment* to their originals in *Vertex*. All other methods are mapped to *skip*. Obviously, the invariant of *Vertex*

$$\begin{tikzcd}
y \arrow[r, "T: m_T"] & C(y) \\
x \arrow[u, "f"'] \arrow[r, "S: m_S"'] & {} \arrow[u, "f"']
\end{tikzcd}$$

Figure 20 Conformance

(which is *true* as it is not defined) is strengthened in *Node*. The above diagram commutes because *environment* is never changed in both *Vertex* and *Node*. Therefore, a node can be used whenever a vertex is expected. The same holds for handles and vertices.

To demonstrate implementational conformance, we present implementation classes for *Hypertext*, *Vertex*, *Node*, and *Handle* in Figures 21, 22, 23, and 24.

```
class Vertex_Implementation
        private     environment: NodeId
                    anc, succ: Set [VertexId]
                    env (): nid: NodeId is nid := environment
end Vertex
```

Figure 22 Implementation class ‘Vertex_Implementation’

```
class Node_Implementation
inherit Vertex_Implementation, conform Vertex_implementation
...
```

Figure 23 Implementation class ‘Node_Implementation’

```
class Handle_Implementation
inherit Vertex_Implementation, conform Vertex_implementation
...
```

Figure 24 Implementation class ‘Handle_Implementation’

```
class Hypertext_Implementation
        private     nodes: Set [NodeId]
invariant   ∀ vid • vid↑.anc↑.environment ∈ nodes ∧ vid↑.succ↑.environment ∈ nodes ∧
            ∀ vid1, vid2 • vid1↑.anc = vid2 ⇔ vid2↑.succ = vid1

            add_node () new-nid: NodeId is
            new-nid := new (Node_Implementation); nodes := nodes ∪ {new-nid}

            remove_node (old-nid: NodeId) is
            nodes := nodes - {old-nid};
            for all vid in old-nid↑.anc do      vid↑.succ := vid↑.succ - {old-nid}      end;
            for all vid in old-nid↑.succ do     vid↑.anc := vid↑.anc - {old-nid}        end;
            for all hid in old-nid↑.handles do
                    for all vid in hid↑.anc do  vid↑.succ := vid↑.succ - {hid}          end;
                    for all vid in hid↑.succ do vid↑.anc := vid↑.anc - {hid}            end   end

            add_link (new-lnk: Link)
            pre     new-lnk.source↑.environment ∈ nodes
                    new-lnk.destination↑.environment ∈ nodes is
            new-lnk.source↑.succ := new-lnk.source↑.succ ∪ {new-lnk.destination};
            new-lnk.destination↑.anc := new-lnk.destination↑.anc ∪ {new-lnk.source}

            remove_link (old-lnk: Link) is
            new-lnk.source↑.succ := new-lnk.source↑.succ - {new-lnk.destination};
            new-lnk.destination↑.anc := new-lnk.destination↑.anc - {new-lnk.source}

            links_from (nid: NodeId) from: Set (Link) is
            for all vid in nid↑.succ do         from := from ∪ {(nid, vid)}        end;
            for all hid in nid↑.handles do
                    for all vid in hid↑.succ do from := from ∪ {(hid, vid)}        end   end

            links_to (nid: NodeId) to: Set (Link) is
            for all vid in nid↑.anc do          to := to ∪ {(nid, vid)}            end;
            for all hid in nid↑.handles do
                    for all vid in hid↑.anc do  to := to ∪ {(hid, vid)}            end   end

            init () is nodes := { }

end Hypertext_Implementation
```

Figure 21 Implementation class 'Hypertext_Implementation'

The implementation of the module *Hypertext_Model* includes a change of representation. The class *Hypertext_Implementation* does not supply an attribute *links*. Instead, each vertex is provided with two attributes, *anc* and *succ*, that refer to their ancestors and successors with respect to the link structure. Thus, keeping all links of the hypertext together in a set of links is now redundant. The classes *Node_Implementation* and *Handle_Implementation* are the same as their specifications, with the exception that they inherit from *Vertex_Implementation* and not from *Vertex*. The class *Link* needs no implementation.

In this case, it is impossible to prove for a single class that the implementation conforms to the specification. Since the change of representation affects several classes, the proof obligation is to show that the implementation of the whole module *Hypertext_Model* conforms to its specification.

It is beyond the scope of this paper to give the complete proof. We only give the mapping from the implementation classes to the specification classes which is not as simple as the one before, due to the change of representation.

We map an object of *Hypertext_Implementation* to an object of *Hypertext* as follows:

$$\begin{aligned} f\,(\text{hypertext\_imp}) = {} & (\text{hypertext\_imp.nodes}, \\ & \{(\text{vid.in}, \text{vid}) \mid \text{vid} \in \text{hypertext\_imp.nodes} \vee \\ & \qquad \exists\, \text{nid} \in \text{hypertext\_imp.nodes} \bullet \text{vid} \in \text{nid.handles}\}) \end{aligned}$$

Each implementation method is mapped to the specification method with the same name.

In this case, the diagram in Figure 20 commutes. The informal understanding is the following. We assume that there is an abstract hypertext object according to the specification and a concrete hypertext object instantiated by the implementation class. If applying a method of the specification method to the abstract object yields a certain result, then we can ensure that applying the corresponding implementation method to the concrete object yields the same result. (The result is described by a predicate on the state space of the specification.)

Thus we can use a concrete hypertext object when an object as described by the specification class *Hypertext* is expected. For instance, classes from the module *Hypertext_Controller* will probably expect hypertexts according to their specification. In this case, concrete *Hypertext* objects will do.

5 Concluding Remarks

The aim of this case study was to demonstrate formal object-oriented software development. We have applied Responsibility-Driven Design to the development of a hypertext system, we have derived a formal object-oriented specification from the design model, and we have demonstrated the verification of both the consistency of the specification and the correctness of an implementation with respect to the specification.

We can draw several conclusions from this case study.

- Responsibility Driven Design is an excellent method for object-oriented modelling with regard to formal object-oriented software development. The method provides techniques to discover classes, responsibilities, and collaborations. Introducing responsibilities into the design allows for precise interfaces classes. Contracts of the design turn out to be public methods in the specification. Thus, a model according to Responsibility-Driven Design can be appropriately transformed into a formal specification.
- An even closer integration of design techniques and specification techniques is a good starting point for future work. A more precise understanding of different class relationships at the design stage (based on their semantics in the specification) is worth further investigation.
- The specifics of object-oriented programming can cause problems when formal methods are applied to them. In particular, object identities are not easy to deal with. However, the concept of introducing modules to provide the global scope for object identifiers makes the situation manageable. Objects can be referenced via their identifiers, which is typical of object-orientation, and modular verification is nevertheless possible.

- Consistency proofs are comparatively easy to handle. The proof techniques we have applied in the case study provide an appropriate means to perform such proofs. However, conformance proofs require a much larger effort, particularly if conformance between whole modules needs to be proven. Proofs of such an extent typically require machine support.
- Modules play an important role in the specification. First, they are necessary to deal with object identities and second, conformance often holds only between modules and not between single classes. Furthermore, invariants often refer to objects of different classes and are thus better kept in a module than in a class (cf. [13]). Future work will also include the development of a more precise understanding of modules, their interfaces, and relationships between modules.

Acknowledgements

Several colleagues made valuable contributions to this paper. Thanks are due to Eduardo Casais, Claus Lewerentz, Thomas Lindner, and Franz Weber for fruitful discussions, and particularly to Emil Sekerinski whose ideas and suggestions had a considerable and profitable influence on the development of this paper.

This work has been supported by the German Ministry of Research and Technology (BMFT), grant number ITS 9002 M, project KorSo.

References

[1] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, 1991.

[2] C.A.R. Hoare. *Proof of Correctness of Data Representations*, in Acta Informatica No. 1, pp. 270-281. Springer Verlag, 1972.

[3] E. W. Dijkstra (Ed.). *Formal Development of Programs and Proofs.* Addison-Wesley, 1990.

[4] A. Goldberg, D. Robson. *Smalltalk-80 The Language and its Implementation.* Addison Wesley, 1983.

[5] Danny B. Lange. *A Formal Approach to Hypertext Using Post-Prototype Formal Specification*, in D. Bjørner, C.A.R. Hoare, H. Langmaack (Eds.), *VDM '90 - VDM and Z - Formal Methods in Software Development*, pp. 99-119. Lecture Notes in Computer Science (LNCS) No. 428. Springer Verlag, 1990.

[6] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[7] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[8] Andreas Rüping, Franz Weber, Walter Zimmer. *Demonstrating Coherent Design - A Data Structure Catalogue*, in Proceedings of *TOOLS USA '93 - Technology of Object-Oriented Languages and Systems*. Prentice Hall. To appear.

[9] Andreas Rüping. *Coffer: Methodology and Language*. Technical Report, Forschungszentrum Informatik Karlsruhe. To appear.

[10] Markku Sakkinen. *Disciplined Inheritance*, in Stephen Cook (Ed.), *ECOOP ‘89 - European Conference on Object-Oriented Programming*, pp. 39-56. British Computer Society Workshop Series. Cambridge University Press, 1989.

[11] Emil Sekerinski. *A Calculus for Predicative Programming*, in R. S. Bird, C. C. Morgan, J. C. P. Woodcock (Eds.), *Mathematics of Program Construction*. Lecture Notes in Computer Science (LNCS) No. 669. Springer Verlag, 1993.

[12] Emil Sekerinski. *A Behavioral View of Object Types*. Technical Report 7-93, Forschungszentrum Informatik Karlsruhe, 1992.

[13] Clemens A. Szyperski. *Import Is not Inheritance - Why We Need Both: Modules and Classes*, in Ole Lehrmann Madsen (Ed.), *ECOOP ‘92 - European Conference on Object-Oriented Programming*, pp. 19-32. Lecture Notes in Computer Science (LNCS) No. 615. Springer Verlag, 1992.

[14] Franz Weber. *Getting Class Correctness and System Correctness Equivalent - How to Get Covariance Right*, in Raimund Ege, Madhu Singh, Bertrand Meyer (Eds.), *TOOLS 8 - Technology of Object-Oriented Languages and Systems*, pp. 199-213. Prentice Hall, 1992.

[15] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.