Extending UML to Model Hypermedia and Distributed Systems*

Luis Mandel
Forschungsinstitut für Angewandte
Software Technologie (FAST e. V.†)
Arabellastr. 17
D-81925 München, Germany
Tel: +49 89 920047 39
Fax: +49 89 920047 18
mandel@fast.de

Nora Koch, Christoph Maier
Institut für Informatik‡
Ludwig-Maximilians-Universität München
Oettingenstr. 67
D-80538 München, Germany
Tel: +49 89 2178 2177
Fax: +49 89 2178 2152
{kochn,cmaier}@informatik.uni-muenchen.de

4th February 1999

Abstract

The present report presents conservative extensions to UML in order to model distributed aspects and hypermedia systems. Ideas and notation coming from the `YAON` project, the EPK-fix project, a methodology for the development of Electronic Product Catalogs (EPCs), and `OOHDM` –short for "Object-Oriented Hypermedia Design Method"– have been used. `YAON` is a notation designed to graphically document the implementation decisions embodied in object-oriented programs running in distributed systems. `OOHDM` is a methodology for modeling hypermedia applications suitable to model special aspects such as navigation and user interface. This extension provides the user with all the power of UML plus the capabilities of the `YAON` with their locations for the physical encapsulation, the client/server relationship, and some new class, modifiers as well as the navigation among these abstractions levels modeled using the `OOHDM`/EPK-fix approach.
Keywords: Modeling Language, Object-oriented Design, Distributed Systems, Object Notation, Multimedia, Hypermedia.

*This work was partially supported by the Bayerische Forschungsstiftung.

1 Introduction

The explosion of Internet applications, specially WWW applications with all their multimedia aspects such as the combination of text, hypertext, images, computer animations, video, sound has raised the necessity of formal or semi-formal methodologies for development such applications. Typically a Web application running on the Internet is based on a client/server architecture and commonly has many distributed aspects. HTML pages as well as applications (i.e. cgi scripts, java applets, etc.) can be distributed among different servers.

In general notations like OMT [Rum95] or even the de facto standard UML [RAT97] do not cover the requirements for the formalisation and graphic documentation of hypermedia and distributed systems described above.

This paper aims at presenting an extension of UML to cover this gap. For the distributed aspects different notations like MOSES [RHSL96a, RHSL96b], ION [AI95], yaon [MM97], etc. has been researched and mostly of the notation come from the `YAON` mainly because it was easier to adapt to the UML notation. `YAON`, short for Yet Another Object Notation, is a practical notation designed to graphically document the implementation decisions embodied in object-oriented programs running in distributed systems and open networks using different communication protocols derived from the *Intermediate Object Notation* (ION) (see [AI95]) developed at the NASA by Colin Atkinson and Michel Izygon. New concepts have been added in order to specify how systems running in a physical location communicate with other entities physically located in other places. In addition, there are defined some new concepts such as synchronized and active classes not present in UML.

For modeling hypermedia systems ideas, notation, and concepts coming `OOHDM` (see [Sch97]) and EPK-fix (see [KKW+97]) have been taken on.

`OOHDM` (Object-Oriented Hypermedia Design Method) uses abstraction and composition mechanisms in an object-oriented framework to allow, on one hand, a concise description of complex information systems, and on the other hand, to permit the specification of complex navigation patterns and interface transformations. In `OOHDM`, a hypermedia application is built in a four-step process supporting an incremental and prototyping process model. Each step focuses on a particular design concern and builds an object-oriented model. These steps are the *Conceptual Design*, *Navigational Design*, *Abstract Interface Design* and the *Implementation*.

In EPK-fix, a catalogue (viewed as a hypermedia application) comprise four interacting components called *structure*, *layout*, *direction* and *services*. An extra interactive component is an existing *product database*. With *structure* is meant the skeleton of the hypermedia application, *layout* is the component which comprises the static aspects, i.e. frames, windows, etc. of the application whereas *direction* is the component where the navigational aspects are described. It is divided into macro and micro navigation. The first one is the navigation through the catalogue while the second one is the navigation inside a frame or a window. *Services* add some comfort to the application, such as help methods, search methods, and bindings to scripts.

The resulting notation has been made in an orthogonal way. The amalgamation of the extensions for modeling distributed systems and the ideas and notation coming from the OOHDM/EPK-fix approach for multimedia and hypermedia modeling, brings the user a natural way to describe a system both in its structure and its user interface showing *which* information is going to be shown and *how* these information is going to be navigated.

Outline:

This document is structured as follows. In Section 2, the origin and the intention of UML is explained, where in sect. 3 the UML extensions for distributed systems are given. These extensions are going to be used in the sect. 4 where the proposed extension for hypermedia is done. Finally, some conclusions and future research lines are given.

2 The Unified Modeling Language UML

The Unified Modeling Language (UML) is the synthesis of many notations developed for software engineering and object-oriented analysis and design methods appeared in the last twenty years. It has been developed by Booch, Rumbaugh and Jacobson, unifying concepts and ideas coming from OMT [Rum95], the use cases from Jacobson [BC89], the CRC (Class-Responsibility-Collaboration) cards from the smalltalk community [WBWW90], the statecharts from Harel [Har87], and many other were introduced to UML and now this unified notation is a de facto industrial standard which has been also aproved by the standarisation comitte of the OMG (Object Management Group).

The result is a modeling language (and *not* a methodology) for specifying, visualizing, constructing and documenting the artifacts of a system intensive process. Most methods consists of both a modeling language and a process. The modeling language is the (mainly graphical) notation that methods use to express design where the process is their advice on what steps are to be taken in doing a design.

3 Adding Extensions for Distributed Systems

In this section, the extensions to UML for distributed systems are presented. These extensions are based on the YAON notation. The goal of YAON is to document graphically implementation decisions of distributed systems with a strong bias to the Java language. Therefore some Java-like class, method and attribute modifiers have been added to the notation. Nodes representing classes are extended to specify active and synchronized classes. Basically a new relationship between classes is added: the client/server relationship. The nodes and arcs are described in the following Section.

3.1 Classes

Two new types of classes are supported using the UL stereotypes. They are the ≪*active*≫ and ≪*synchronized*≫ classes. The class system is restricted to be flat, i.e. there is no way to declare local classes inside a class nor anonymous classes. The basic class icon is presented in Fig. 1:

standard_class
Attribute_1 Attribute_2 ...
Method_1() Method_2() ...

Figure 1: Standard Class

The class name, as always, is written on the top part of the rectangle as indicated. The attributes of the class are described in the middle section of the icon, and the methods of the class in the lower section. This general arrangement of information is retained wherever properties of classes or methods are defined. The ellipses (...) appearing after the attributes and methods are significant, and are used to indicate that an annotation only contains partial information. For example, if a class has exactly two methods, this is indicated by the list *method_1(), method_2()*. If the class has more methods beside these two, this is indicated by *method_1(), method_2(),* ... . Thus it is possible to derive from the diagram whether all the information has been included or whether some is missing. As a shortcut, the attributes and methods may be omitted.

The following visibility modifiers can be specified: public, private, protected and default. These modifiers are those used in the Java programming language and will be specified as a tagged value (i.e. as a metamodel attribute) between braces under the class name and as a prefix of the method or attribute name where their corresponding symbols are shown in Fig. 2: If no modifier is used, the default visibility is assumed.

Modifier	Symbol
public	+
private	-
protected	#
default	

Modifier	Class	Method	Attribute
final	*	*	*
native		*	
synchronized	*	*	
transient			*
volatile			*
abstract	*	*	
static	*	*	*

Figure 2: Visibility and Special Modifiers

Also special modifiers applicable to classes, methods and attributes are available. These special modifiers have no representation in UML and therefore they will be used as *property strings* of attributes and methods with the standard representation of UML, i.e. between braces, as the tagged values of classes. Fig. 2 gives an overview. The modifiers can be combined as for example:

+ <u>main</u> (String [] args)

This specifies the public static method main of a given class. An example, which shows the use of modifiers, is depicted in Fig. 3.

Default values as well as type (or class) of attributes can be specified. The = notation will be used for that purpose. For example, if an attribute called "counter" should be an integer whose initial value is 1 this will be specified as "counter: Integer=1". If the attribute "counter"' can have only values $i_1, \ldots, i_n$ then those values will be specified using an extensive comma list containing all the possible values, i.e. as "counter: Integer=$[i_1, \ldots, i_n]$".

An active class is one whose instances have an independent thread of control. Thus, they have the ability to run methods "spontaneously". In contrast, an instance of a passive class executes a method only when it is called by another object. After completing the method it is passive again. To depict an active class, the basic class icon is extended with the stereotype as shown in Fig. 3.

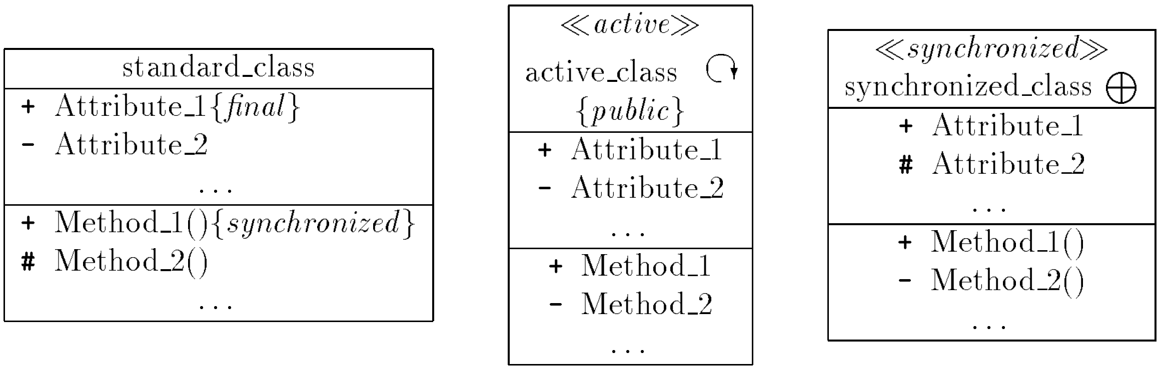

Figure 3: Standard, Active and Synchronized Classes Using Modifiers

Synchronized classes are those whose methods are protected from concurrent invocation of their methods. A circle with two crossed lines inside has be chosen to annotate the stereotype for synchronized classes (Fig. 3). Some "synchronisation protocol" exists in order to protect them. The synchronization of the methods is also described in the class box. Icons for the most common synchronization protocols, such as mutual exclusion, readers/writers, guarded are provided. Actually, the protocol icon must be interpreted as a shorthand of another diagram where the synchronization protocol is specified. These protocols can be specified using Petri nets, state-charts, etc. If the user would like to have a new protocol, he has to chose a new icon and must give the protocol specification using one of the formalisms mentioned above. Examples of synchronized classes are given in [MM97].

3.2 Clientship Relation

A clientship relation exists between a class called *client* and a class called *server*. This relation is unidirectional and there exists an arrow from the client_class to the server_class if the client *uses* a method from the server. This is a static relation denoted by the stereotype ≪*client/server*≫, and then it can be detected at compilation time (see Fig. 4).

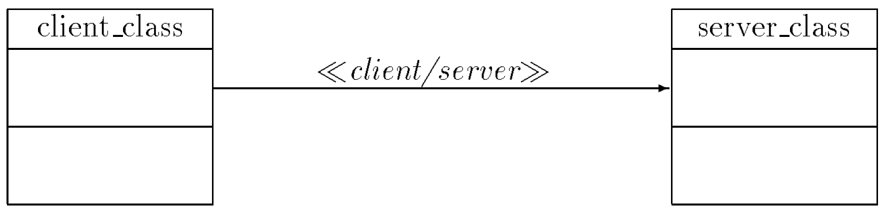

Figure 4: Client Server Relationship

In the following different stereotypes for the Client/Server relationships are presented.

Permanent versus Transient Relationships

If the client has the data in its main structure, the relation is said to be *permanent*. Otherwise, if the client objects have visibility only during the method call, the relation is said to be *transient*. A permanent relation is denoted with a circle inside the class symbol and by the stereotype ≪*perm*≫; a transient relation with a circle outside the class symbol and the stereotype ≪*trans*≫ (see Fig. 5). Using these relationships, one can specify the interval of time a relation exists.

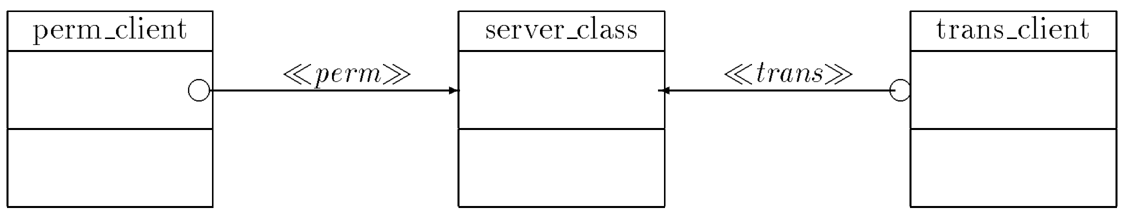

Figure 5: Permanent versus Transient Relationships

Attached versus Detached Relationships

It can also be distinguished whether the client holds a reference to the server or if it holds the actual state of the server (a value). In the first case the relationship is said to be *detached*, whereas in the second case, it is said to be *attached*. A detached relationship is useful in the following situation: A client requests information from the server. This information is stored as an object inside the server. If the server passes a reference to this object, the client can use this reference to change the object inside the server directly. To avoid this, the server must send the value of the object to protect it from unproper use (see [Atk91]).

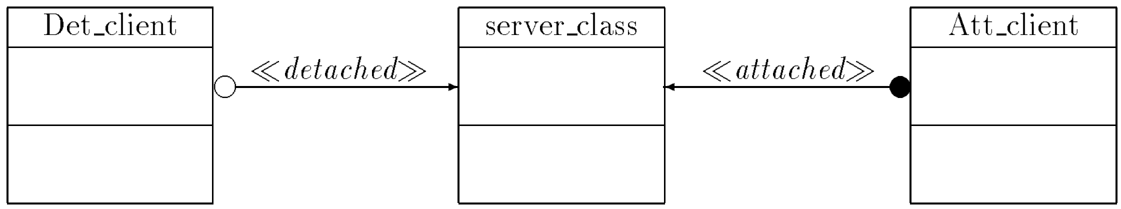

Figure 6: Attached versus Detached Clientship Relation

Every clientship arrow has a small circle at the client's end (i.e. the tail). If the circle is transparent the clientship is detached whereas if the circle is a black bullet the clientship is attached. The server is in a sense a part-of, or stored within it. Also the stereotypes ≪*attached*≫ and ≪*detached*≫ will be added to the respective arcs.

There are in `YAON` more stereotypes for clientship annotations which are optional, such as the stereotype for procurement annotation, stereotype for call annotations, stereotype for asynchronously executed methods, stereotype for reference annotations, stereotype for multiple clientship relation. For more information about these steretypes see [MM97].

3.3 Virtual Nodes

The concept of virtual node is introduced to `YAON` as Atkinson did in [Atk91]. This concept is missing in UML and it is indispensable in order to specify "real" distributed systems. Virtual nodes are clusters of classes (or packages) represented bas bevel-edges.These clusters are the set of nodes/classes which will potentially be distributed to physical locations. In this way remote client/server relationships can be specified. An example of this notation is given in Fig. 7. It includes three virtual nodes: node_1, node_2 and node_3. node_1 only contains the Class_B, node_2 contains the Class_Y and Class_Z and node_3 contains a package Pack_A which contains the Class_X. The Class_B from node_1 is in a client/server relationship with the Class_Y from node_2 and it is also in a client/server relationship with the Class_Z from the same node. The Class_X from package_A from node_3 is also in a

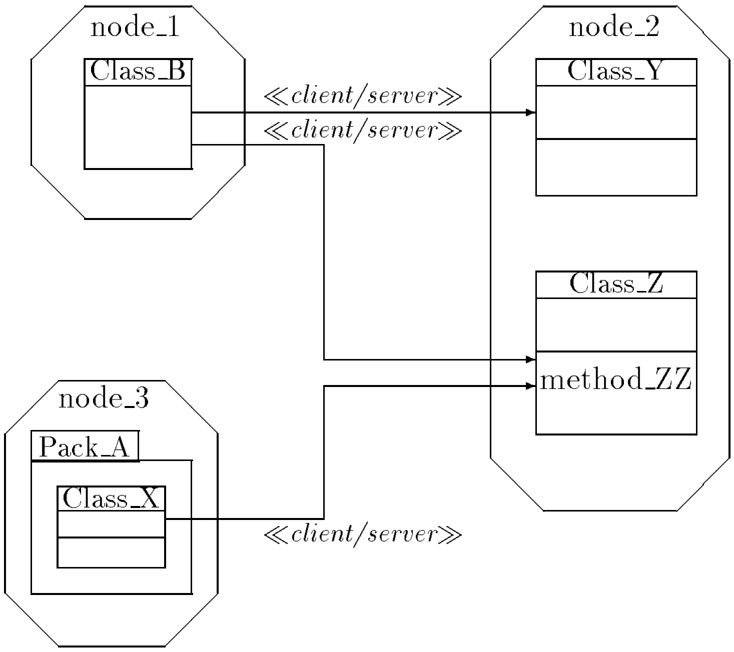

Figure 7: Virtual Nodes

client/server relationship with the Class_Z of node_2. The communication between virtual nodes can be done by normal method invocation as described in 3.2 or, if the virtual nodes are to be distributed, remote communication mechanisms can be used as described in [MM97].

Note that using the client/server relationship and the virtual nodes one can specify distributed multimedia applications with client and server-side applications such as dynamic generated pages via CGI-applications, links to CGI-scripts or servlets running on remote servers, embedded applets. Also synchronization between multimedia objects can be specified, like the specification of a video which should begin when a link is activated.

4 Adding Extensions for Hypermedia

In this section the extensions to UML for multimedia and hypermedia design are presented. These extensions are based on the EPK-fix and `OOHDM` methodologies as described in [KKW+97] and [Sch97] respectively. Basically UML will be extended with navigational modeling features i.e. a graphic notation will be added to UML in order to describe *which objects* and *how* are going to be visited and in which contexts. Also a notation for the *Layout* or *Abstract User Interface* will be given.

For the *Conceptual Design* the extensions already presented in Section 3 will be used.

These extensions are powerful enough for the specification of object-oriented systems and distributed systems such as web applications.

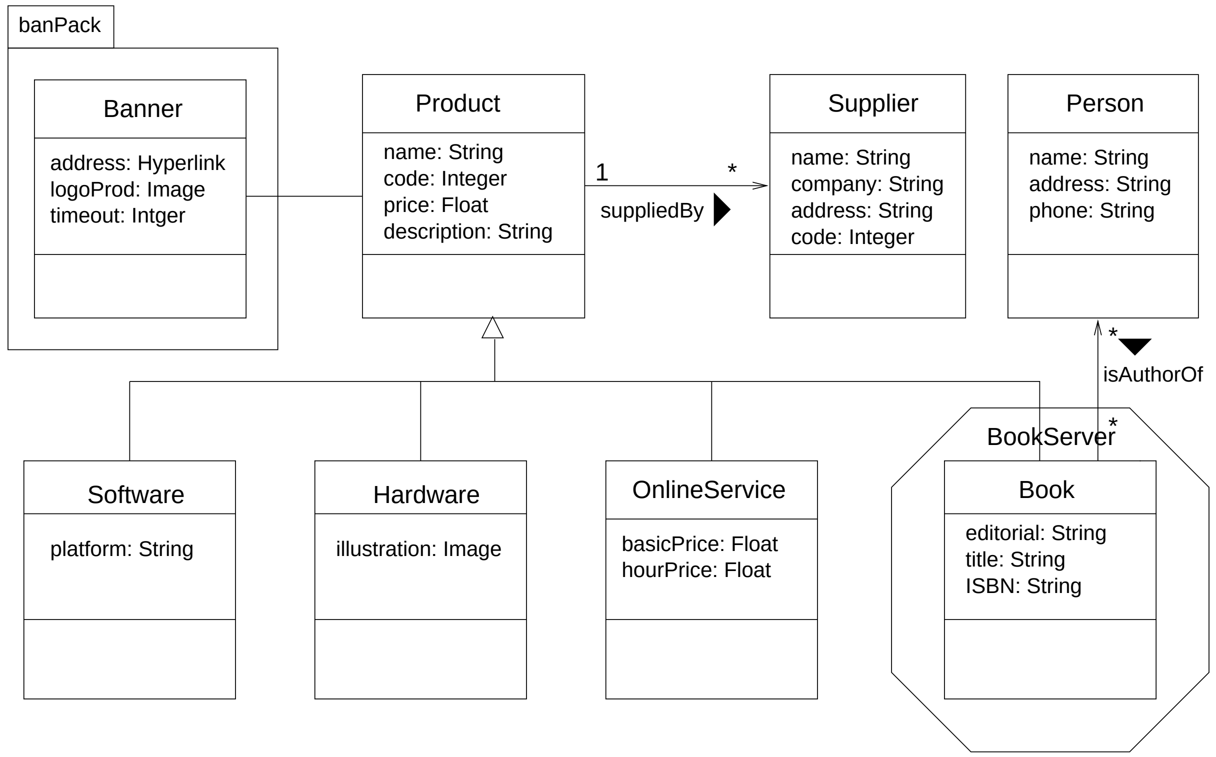

Figure 8: An EPC of a Software-House

Following the approaches presented above we assume that a *Conceptual Design* made of classes and relationships between classes already exists. The presentation of the extensions will be made by using an example of a simplified EPC of a Software House. The conceptual design of such catalogue is depicted in Fig. 8. The class diagram shows the class Product that has four subclasses, namely Software, Hardware, OnlineService and Book. For each product there are many suppliers represented by the Supplier class. Each product has a name, a code, a price and a description. Software products have an attribute platform which is a string representing the platform where the product runs. For Hardware products there is an image of the product, OnlineService have a basic price and an hour price and Book have attributes representing the author, the title and the editorial. Book are encapsulated in a package called BookServer located in a remote server.

The class Banner is a subsystem having an aggregation relationship with the Products class. This banner is actually an application which during an amount of time shows some advertisement in the following way: An image of a given product (of the same category) is displayed and the whole image is a button which when clicked will activate the corresponding hyperlink to that product. After the timeout limit is reached a new hyperlink and a new timeout is set and a new image corresponding to another product (of the same category) will be visible. This is done in a cyclical way, that is, when the last product is reached the banner application will begin with the first product again.

We begin the description of the extensions with the *Direction* or *Navigational Design*, which is followed by *Abstract Interface Design* or *Layout*.

4.1 Notation for the Navigation

Navigation is constructed as a view (in the database way) over the conceptual model or structure. The navigational model is obtained from the conceptual model through elimination or addition of classes as well as the definition of new attributes and relationships.

In `OOHDM`, navigational design is expressed in two schemas: The navigational class schema and the navigational context schema. Navigable objects of an application are defined by classes in a navigational class schema, whose classes reflect the chosen view over the application domain. In `OOHDM` there is a set of predefined types of navigational classes: nodes, links, contexts and access structures. The semantic of nodes and links is the natural one in hypermedia applications whereas access structures, such as indexes represent possible ways of accessing nodes.

A navigational object, also called navigational node or simply node is an instance of a class in the navigational class schema.

Three new attributes have been added here to the navigational objects. They are the *refresh* of type link and the *timeout* of type integer. They work together and the intended semantics is that the current navigational object will load the link specified by the attribute *refresh* after the time specified by the *timeout* attribute is reached. If no *timeout* is specified an infinite amount of time will be assumed. The link *refresh* can have a special value which is *self*. In this case the same navigational object will be reloaded after the timeout is reached. This is special useful for navigational objects which have information dynamically generated, like scoreboards.

Yet another new attribute has been included and it is the *expiration* one, which is of type date. The intended semantics is that the information contained by that navigational object will expire after the date specified by this attribute. Specific browser can use this attribute in order to display with special layout (or not display at all) the expired information. User specific attributes can also be added, like *name* of the navigational object, *language*, *author*, *keywords*, etc.

In an analogous way, links reflect relationships, as already said, intended to be explored by the final user and are also defined as views on relationships in the conceptual schema.

The general syntax for defining the attributes of navigational objects is shown in Fig. 9, where:

- *nodeName* is the name of the class of nodes we are creating.

NODE *nodeName* [FROM *className*:*varName*] [INHERITS FROM *nodeClass*]
$attribute_1$:$type_1$ = [SELECT $name_1$]
[FROM $class_1$:$varName_1$, ... $class_j$:$varName_j$
[WHERE *logical-expression*]
$attribute_2$:$type_2$ = [SELECT $name_2$] ...
...
$attribute_n$:$type_n$ = [idem]
refresh:*Link*
timeout:*Integer*
expires:*Date*
END

Figure 9: Node Definition

- *className* is the name of a conceptual class (from which the node is being mapped);
- *nodeClass* is the name of the super-class;
- $attribute_i$ are the names of attributes for that class, $type_i$ the attribute's types;
- $name_i$ are the subjects for the query expression and $varName_i$ are existentially quantified variables used to express logical conditions;
- *logical-expression* allows defining classes whose instances are a combination of objects defined in the conceptual schema when certain conditions on their attributes and/or relationships hold;
- *timeout* is an integer specifying the amount of time to be waited in order to load the node specified by the attribute *refresh*;
- *refresh* is a link to a node. A special value *self* is allowed which is equivalent to a link to itself;
- *expires* is a date specifying the expiration of the information contained by the current node.

As an example, consider the EPC of Fig. 8. In addition the navigational class schema will be designed (see Fig. 10).

Note that in this "view" Person has been eliminated as a class and that information about persons will be added to the class Book via an sql-like query. The right-hand-side of the query (author) is a new attribute of the Book class whereas the left-hand-side is a set whose cardinality is given by the cardinality of the relation in the conceptual design. The WHERE clause of the query uses the name of the relationship isAuthorOf for determining which are the authors of the given book. Similarly for the Supplier class.

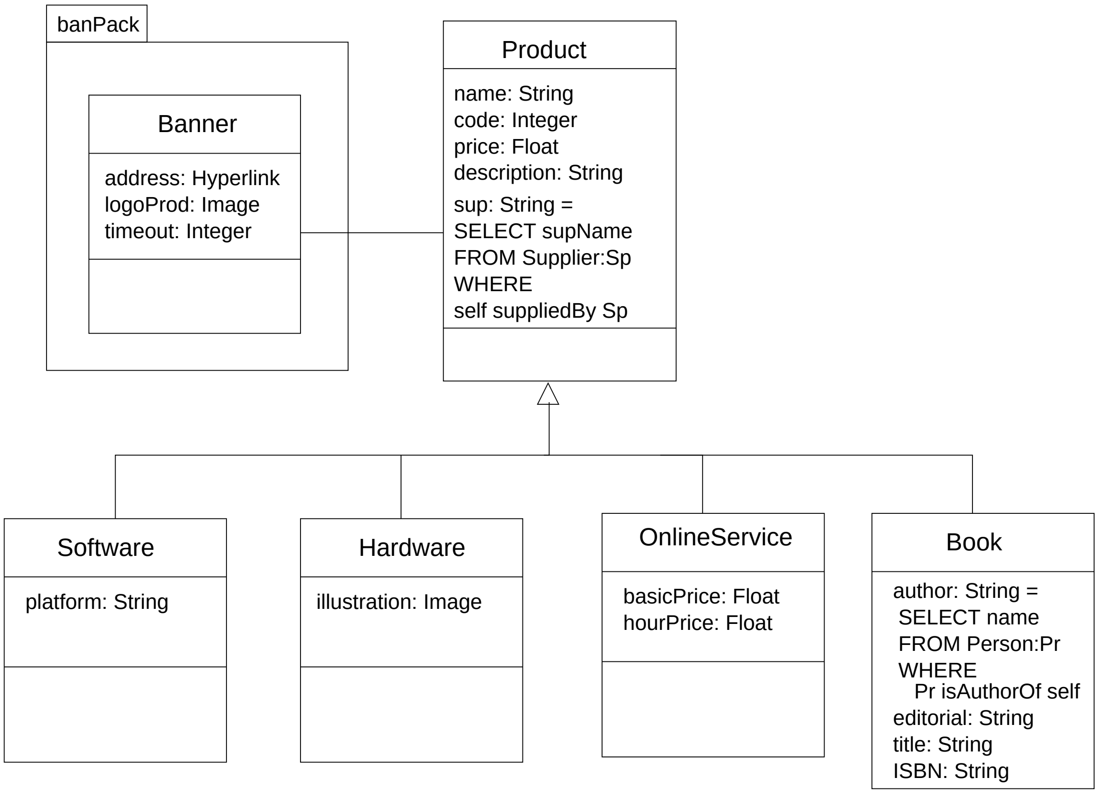

Figure 10: Navigational Class Schema of an EPC of a Software-House

On the other hand the class Banner still appears in the navigational class schema having an aggregation relationship to Product. This class is treated as a subsystem and therefore its navigation will be defined in its own navigational class schema.

Up to now the navigation was possible only between single objects but navigation within set of navigational objects and between sets of navigational objects is needed. To address this problem the "navigational contexts" have been defined. Normally such information are back-links, class hierarchies, menues, etc. In OOHDM, the main structuring primitive of the navigational space is the notion of navigational context . A navigational context is a set of navigational objects and other (nested) navigational contexts. It may be defined intensionally or extensionally, by either defining a property that all nodes and links in the context possess, or by enumerating its members. The definition of a context also includes a traversal order of its elements, and the existence or not of associated access structures. In this way a new kind of diagram is introduced: The "navigational context schema". Whereas the "navigational class schema" of above shows *which* information is going to be visited the "navigational context schema" shows *how* this information is going to be accessed.

Queries come in two flavours. Normally a query takes a user input and either generates an index with the result or directly goes to the first result node which has a link to the following and so on. Icons for such queries are shown in Fig. 11.

Figure 11: Query and Index Query

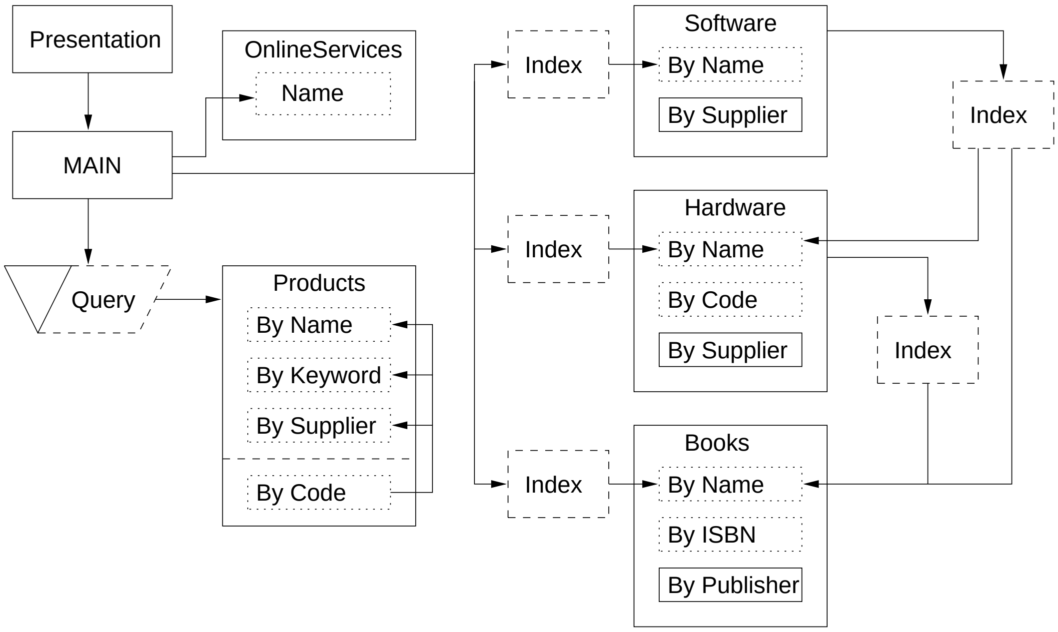

Figure 12: Navigational Context Schema of an EPC of a Software-House

Navigational contexts organize the navigational space into consistent sets that can be traversed following a pre-established order. Fig. 12 shows the navigational context schema of the EPC whose conceptual design has been presented in Fig. 8 and whose navigational class schema has been depicted in Fig. 10. In this schema, it can be observed that there is a Presentation that will be shown only once. No back-navigation is possible to the presentation. A Query possibility is presented and a link to OnlineServices is displayed. The query directly interacts with the navigational node Products, i.e. input queries made by the user will generate a mixed index of products with the matching results. Navigation inside Products can be done from the context Code to any other context but not vice versa. From the MAIN node one can directly reach the OnlineServices node. It is thought that OnlineServices has only a context where the user can access to online services, like encyclopedias, dictionaries. The MAIN node has links to three indexes (Software, Hardware and Book). Using those indexes the user can access the respective products "inside" the Name context. Navigation among contexts defined for the same class is permitted as well as navigation from Software-Products to Hardware-Products and navigation from Software-Products to Book-Products. That is, given a Software-Products the user can access to related Hardware-Products and to related Book-Products. Navigation from Hardware-Products to related Book-Products is also possible but neither navigation from Book-Products to Hardware-Products nor navigation from Book-Products to Software-Products is allowed in this model.

4.2 Notation for the Layout

The Abstract Data Views (ADV) design model was originally created to specify the separation of the user interface from the application components of a software system [CCL93]. This interface can be exercised through messages (in particular external events generated by the user).

ADVs have been conceived to represent interfaces between different media such as networks, users, or as interface among Abstract Data Objects (ADOs). Both have a state given by their attributes and methods or actions which can change or query the state. ADVs have additionally an interface and they are an abstract representation of the behaviour, not the implementation.

Typically for one Abstract Data Object there are defined one or more Abstract Data Views which describe how some aspects of its state is presented to the external world.

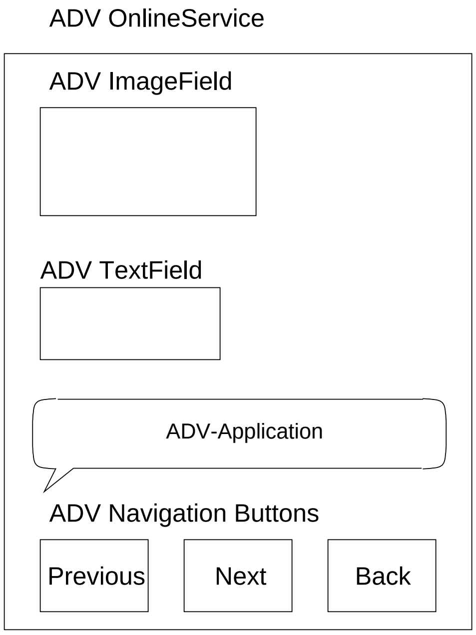

Figure 13: Composition of ADVs

User interfaces can be viewed as a composition of behaviour and structure of simple visual objects. A window, for example, is a composition of menus, dialogue boxes, text fields, images, and applications. Aggregation and inheritance mechanisms are used to define interface perceivable objects as composition of lower-level ADVs and to provide a framework for defining hierarchies of interface objects, respectively. Fig. 13 is an example of structural and behavioural nesting. Static and dynamic aspects of the abstract interfaces are presented graphically in configuration diagrams and ADV Charts respectively.

4.2.1 Configuration Diagrams

Coleman [CHB92] defines configuration diagrams in the context of Objectcharts. They are introduced in the ADV model to specify communication between the data views and the data objects, thus ADVs provide services to the user and require services from the ADO. If the Data View is an interface to the user, it also will receive stimuli from the user in form of mouse click, mouse focus or keyboard input (Fig. 14).

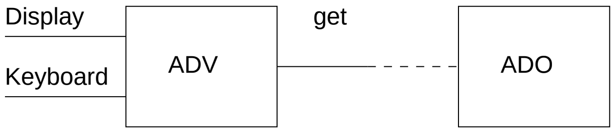

Figure 14: General Configuration Diagram of an ADV

The configuration diagram shows also the composite structure and composite behaviour of the user interface. `OOHDM` uses the ADV approach and has improved the original notation [CHB92] as follows:

- A dashed line is used to permit representation of relationship between enclosed ADVs, e.g. user mouse click on Show related Books will display the ADV ADVBook associated with the corresponding navigational context (see Fig. 15). As result of this invocation a list of bools is displayed. Optional elements are indicated with an o in the right corner.
- Objects that are not reactive to user's activities are defined as ADV's attributes.
- Composition of type AND and XOR are represented by placing objects side by side and lightly superposed respectively.

`OOHDM` proposes to use the size of the included ADVs to indicate relative importance and use this in the implementation phase. We think that such decisions are to be taken in the implementation as they may be dependent of the chosen medium, e.g. images or videos are central point of attention in a hypermedia application on CD-ROM while in the same application implemented for the Web performance aspects have to be considered.

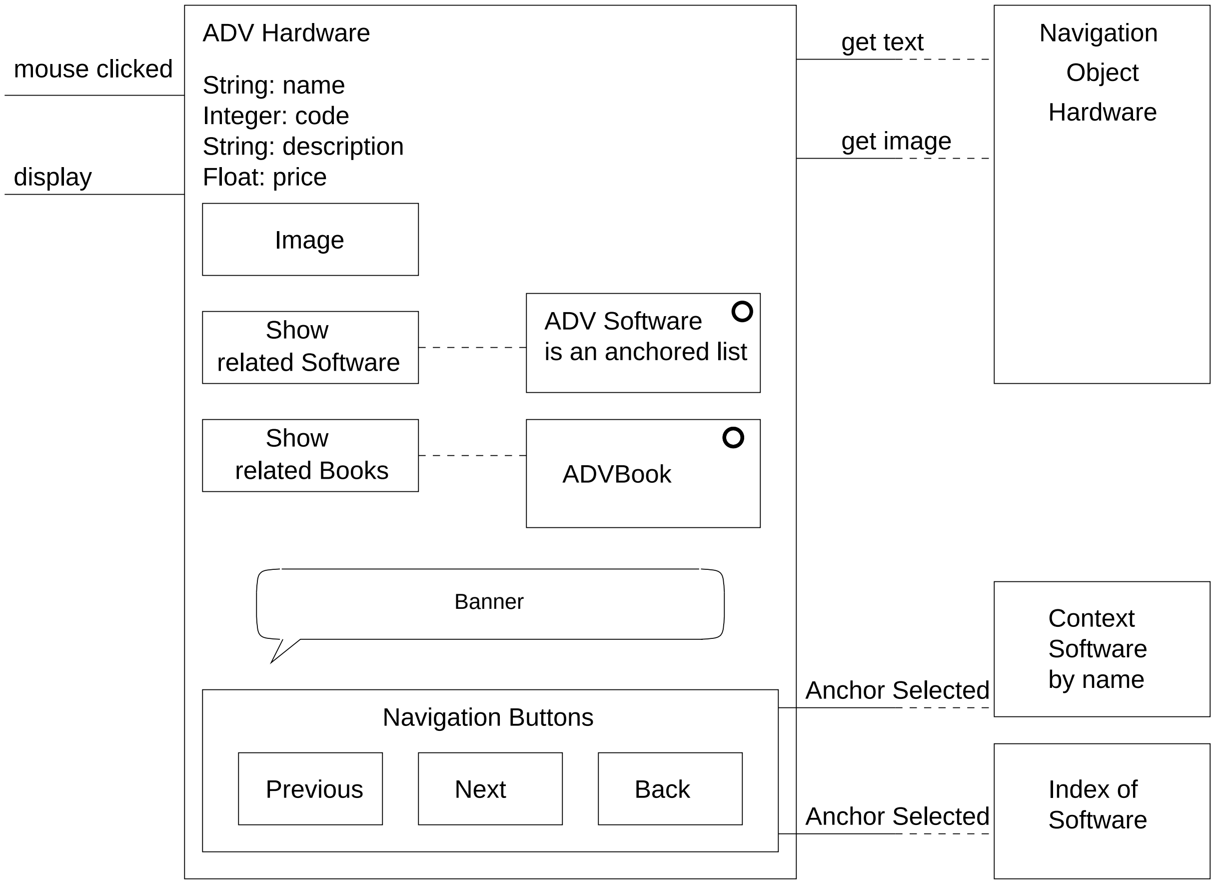

Figure 15: Configuration Diagram for ADV Project

Neither size, position nor other layout characteristics are specified in the ADV approach design. Therefore, ADV objects that are placed side by side in the configuration diagram may be placed in the same place in implementation, i.e. an anchor hidden by a video and with the reaction capability to start the video on mouse click.

A configuration diagram for the ADV Hardware of the Software-House example using the `OOHDM` notation is presented in Fig. 15. Two kinds of reaction to user's actions can be observed. The first type only produces changes in the content and layout of the same composite ADV. The second one is of type navigation, in which case a hyperlink is followed reaching other nodes of the hyperspace. In the EPK-fix methodology [KS97] these two types of reaction are described as micro-direction and macro-direction.

4.2.2 ADVcharts

ADVcharts provide a visual schema for the specification of the dynamic aspects of the user interface. They contain one or more states and transitions as well as may contain attributes and other ADVs to describe their behaviour. They are an extension of Statecharts [Har87] and Objectcharts [CHB92] supporting nesting of states and ADVs. Nesting of states expresses behavioural nesting while nesting of ADVs is the expression of structural nesting. ADVcharts also use notation from VDM and Petri-nets.

As in Statecharts the different states are represented with boxes with rounded corners and the transitions between states with arrows from one state to another. The transition is specified by four fields: transition's name, pre-condition, event and post-condition. The conditions that must be satisfied to fire the transition are given in the pre-condition. The event-field specifies the event that will fire the transition (e.g. mouse click). The post-condition gives the conditions that hold once the transition is fired (Fig. 16).

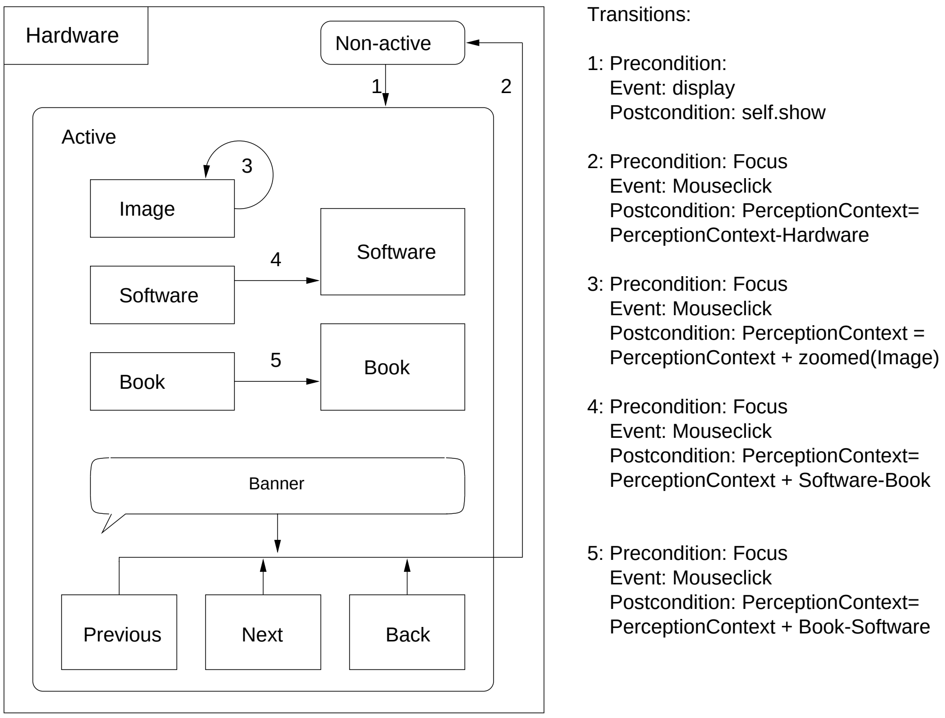

Figure 16: ADV Chart for Project

ADVs allow the representation of synchronisation among different objects using the symbol for synchronisation of Petri-nets ($\bigoplus$) to join those transitions that must be synchronised.

4.2.3 Improvements to the ADV Notation

Although the ADV configuration diagrams of `OOHDM` are more intuitive than the original ones, we consider that additional improvements are required. For certain frequently used user interface objects, such as anchors or input fields as well as for lists of objects a special notation is helpful, even in complex configuration diagrams needed.

Not only users may generate events but also the system. For example timeout events, like the scoreboard of a live match or the rate of the stock market.

In the following a specific notation for User Input, Collections, Anchors, Applications, Sound and System Events are presented.

User Input

Hypermedia applications are changing from being totally passive applications, which offer to the user only the possibilities to read, contemplate and take decisions about which link to follow next, to more active ones. Interactive applications additionally give the users the possibility to query databases, to select and to store data.

We define a notation for ADV input, specifying which information the users are requested to supply. Notice that there is no indication made, if the users are using the keyboard to entry the data or selecting options from a browser or a checkbox. These are implementation aspects to be decided in the corresponding implementation phase. The semantics of this ADV includes the display of the ADV content, the waiting for the user activity, the evaluation of the input and the trigger of the defined event. The graphical notation for the ADV input is a dotted lined box as shown in Fig. 17.

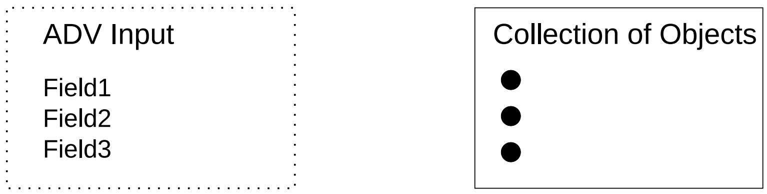

Figure 17: ADV for User Input and Collection of Objects

Collections

A composite ADV contains a list of anchors, a set of options or a list of other interactive ADVs. To avoid the textual description by comprehension or by extension of this kind of composite we introduce a box with a special notation for collections (shown in Fig. 17). If the list is of inputs, the box will be dotted. It is not specified if the list will be horizontal or vertical, objects may even be arranged as a table. In case of a collection of anchors, a navigation path for each object has to be defined.

Anchors

Anchors are the start points of the navigation. There are seldom presented in the literature as independent objects [CB97, HS94] mostly as part of hyperlinks. They comprise at least a presentation and an associated hyperlink. The presentation may be either a text (even single character), an image, a video, a group of mixed media, a special interactive object like a button, or a whole document. In `OOHDM` they are specified as classes that inherit from class text, image, video, etc. and referenced by anchored text, anchored image, anchored video. Our purpose was to find an intuitive notation for anchors since they are one of the most frequently used objects in the hyperspace model, thus we use an underlined description in analogy to the usually notation used by browsers (see Fig. 18).

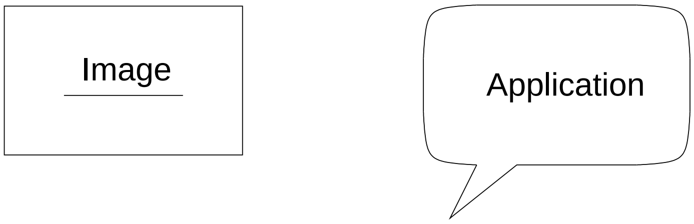

Figure 18: An anchor and an Application Ballot

Applications

Applications are incorporated to hypermedia nodes with increasing frequency. This is the case of applets or embeded objects in WWW. It makes sense to distinguish them with a special notation. We choose a balloon as shown in Fig. 18. Applications can also have navigation, e.g. an application could display a hyperlink. Applications are modeled as subsystems in the conceptual design and in case that an application has a navigational object then it will appear also in the navigational design.

Sound

A special notation for embeded sound has been added to the proposed extension. We choose a speaker to represent that an user interface object has associated a sound file (Fig. 19). This sound file will be loaded by a sound player running in background. Additional information can be specified with notation, i.e. to indicate autostart by node loading a black triangle is used and cyclical play with a loop.

System Events

Changes between states can be the result of internal or external events. Internal events are generated by the transitions. We consider that external events may be caused by the system as well by the user. External events generated by the user are mouse over, mouse click or double click, before, during and after click. External events generated by

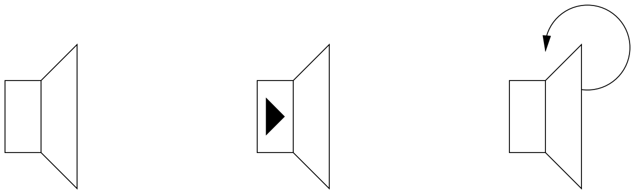

Figure 19: Sound Icons

the system are timeout or refresh for example. They are represented in same way as user events.

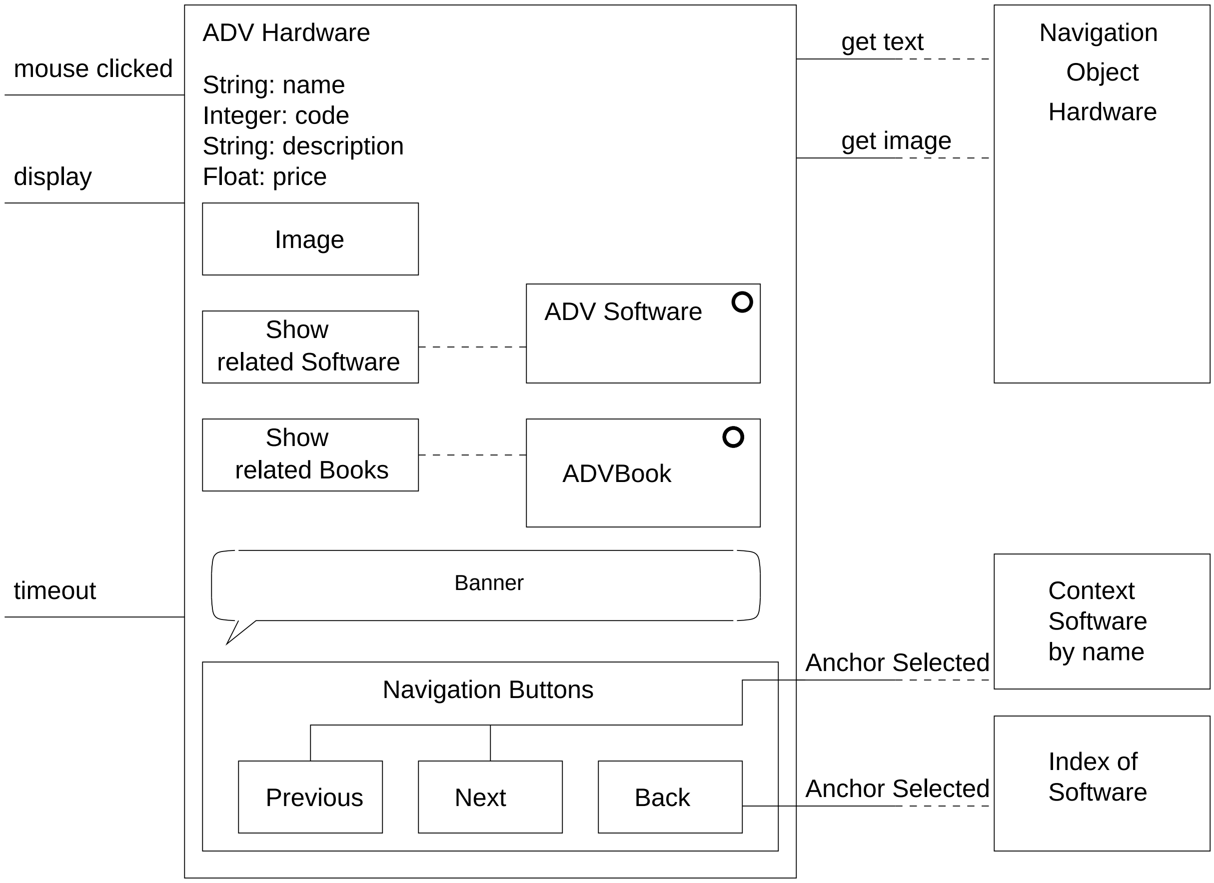

Figure 20: Improved Configuration Diagram

Fig. 20 shows the ADV for the Hardware class of the example of the EPC Software-House using the improved notation.

5 Conclusions and Ongoing Work

An extension to the UML notation, useful for modeling multimedia and distributed aspects of computation has been presented. The notation includes primitives for static definitions of classes, packages and virtual nodes. Classes can have different flavors. They can be declared as standard, active, final, abstract and synchronized. Also visibility class modifiers such as public, private and protected can be used in order to declare a class. Methods and attributes can be declared with the same semantics as in Java. That is, it is supported the same visibility and special modifiers for classes, methods and instance variables as the Java programming language does.

The client/server relationship has been defined and different variants of this relationship are supported. A variant of this relationship can be further detailed with annotations. For example, for a remote client/server relationship it can be specified if the communication will be done using sockets or if it is intended to be done using Remote Method Invocation.

Multimedia aspects have been added to UML in these extensions by using ideas and notation coming from `OOHDM` and EPK-fix. Given a conceptual design of a system a navigational design is generated by adding some `sql`-like queries in order to generate the nodes to be navigated. The Navigational Class Schema shows *which* information is going to be visited whereass the Navigational Context Schema *how* those nodes are going to be navigated.

Finally, using Abstract Data Views, it is shown how the abstract user interface is composed. In this way a web site can be modelled describing the layout of the `html` pages. This pages may contain applications, like applets in WWW, which will be modeled as subsystems in their conceptual design.

This is a first approach of modelling multimedia systems with UML notation. In a future work it will be ported the whole notation and diagrams to UML in order to have a 100% UML complaint.

Acknowledgements: We thanks Hubert Baumeister and María Victoria Cengarle for the fruitful discussions and useful comments about the subject.

References

[AI95] Colin Atkinson and Michel Izygon. ION A Notation for the Graphical Depiction of Object Oriented Programs. Cooperative agreement ncc 9–30, NASA, July 1995. Available at `http://ricis.cl.uh.edu/atkinson/ion/`.

[Atk91] Colin Atkinson. *Object–Oriented Reuse, Concurrency and Distribution – An Ada–Based Approach*. Addison–Wesley Publishing Company, 1991.

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA'89*, volume 24, 1989.

[CB97] Licia Calvi and Paul De Bra. Improving the Usability of Hypertext Courseware Through Adaptive Linking. In *Proceedings of The Flexible Hypertext Workshop*, 1997.

[CCL93] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. Introducing ADVcharts: A Visual Formalism for Describing Abstract Data Views. Technical report, PUC-Rio, July 1993.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1), 1992.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Sytems. *Science of Computer Programming*, 8(3), 1987.

[HS94] F. Halasz and M. Schwarz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2), 1994.

[KKW+97] Alexander Knapp, Nora Koch, Martin Wirsing, Jochen Duckeck, Rainer Lutze, Hartmut Fritzsche, Dietrich Timm, Patrick Closhen, Martin Frisch, Hans-Jürgen Hoffmann, Bernd Gaede, Josef Schneeberger, Herbert Stoyan, and Aandreas Turk. EPK-fix: Methods and Tools for Engineering Electronic Product Catalogues. In R. Steinmetz and L.C. Wolf, editors, *Interactive Distributed Multimedia Systems and Telecommunication Services*, Lecture Notes in Computer Science 1309, pages 199–209. Springer-Verlag Berlin-Heidelberg, September 1997.

[KS97] Nora Koch and Joseph Schneeberger. Integrated Assistance for the Development of Electronic Product Catalogues. In *Proceedings of the Symposium on Software Technology, SADIO*, August 1997.

[MM97] Christoph Maier and Luis Mandel. YAON – a Static Diagram Technique for Object Oriented Distributed Systems. Technical Report 9709, Institut für Informatik der Ludwig-Maximilians-Universität München, 1997.

[Oes97] Bernd Oestereich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, 1997.

[RAT97] RATIONAL Software Corporation. *UML Notation Guide*, September 1997. Version 1.1. Available at `http://www.rational.com/`.

[RHSL96a] G. Rasmussen, B. Henderson-Sellers, and G.C. Low. Extending the MOSES methodology to distributed systems. *Journal of Object Oriented Programming*, pages 39–46, July–August 1996.

[RHSL96b] G. Rasmussen, B. Henderson-Sellers, and G.C. Low. An object–oriented analysis and design notation for distributed systems. *Journal of Object Oriented Programming*, pages 14–27, October 1996.

[Rum95] J. Rumbaugh. Modeling and design – omt: The functional model. *Journal of Object Oriented Programming*, 8(1):10–14, 1995.

[Sch97] Schwabe, Daniel and Rossi, Gustavo. An Object Oriented Approach to Web-Based Application Design. Technical report, Departamento de Informatica, PUC-RIO, Brazil, 1997.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.