10 Model-Based Web Application Development

Gustavo Rossi, Daniel Schwabe

Abstract: In this chapter we present our experience with the Object-Oriented Hypermedia Design Method (OOHDM), a model-based approach for developing Web applications. We first describe the main activities in OOHDM and then we illustrate the application of the method with a simple example, a CD store.

Keywords: OOHDM, Web development, conceptual model, navigation model, hypermedia development.

10.1 The OOHDM approach – An Overview

The Object-Oriented Hypermedia Design Method (OOHDM) is a model-based approach to the development of Web applications. OOHDM uses different abstraction and composition mechanisms in an object oriented framework to, on one hand, allow a concise description of complex information items, and on the other hand, allow the specification of complex navigation patterns and interface transformations. OOHDM provides a clear roadmap that allows answering the following key questions, generally asked when building Web applications:

- What constitutes an "information unit" with respect to navigation?
- How does one establish what are the meaningful links between information units?
- How does one organise the navigation space, i.e., establish the possible sequences of information units the user may navigate through?
- How will navigation operations be distinguished from interface operations and from "data processing" (i.e., application operations)?

In OOHDM, a hypermedia application is built in a five-step process supporting an incremental or prototype process model. Each step focuses on a particular design concern, and an object-oriented model is built. Classification, aggregation and generalisation/specialisation are used throughout the process to enhance abstraction power and reuse opportunities. Table 10.1 summarises the steps, products, mechanisms and design concerns in OOHDM.

Table 10.1. Activities and formalisms in OOHDM

Activities	Products	Formalisms	Mechanisms	Design Concerns
Requirements gathering	Use cases, Annotations	Scenarios; user interaction diagrams; design patterns	Scenario and use case Analysis, Interviews, UID mapping to conceptual model	Capture the stakeholder requirements for the application.
Conceptual design	Classes, subsystems, relationships, attribute perspectives	Object-oriented modelling constructs; design patterns	Classification, aggregation, generalisation and specialisation	Model the semantics of the application domain
Navigational design	Nodes, links, access structures, navigational contexts, navigational transformations	Object-oriented views; object-oriented State charts; context classes; design patterns; user-centred scenarios	Classification, aggregation, generalisation and specialisation.	Takes into account user profile and task. emphasis on cognitive aspects. build the navigational structure of the application
Abstract interface design	Abstract interface objects, responses to external events, interface transformations	Abstract interface widgets; concrete widgets; ontologies; design patterns	Mapping between navigation and perceptible objects	Model perceptible objects, implementing chosen metaphors. Describe interface for navigational objects. Define layout of interface objects
Implementation	Running application	Those supported by the target environment	Those provided by the target environment	Performance, completeness

We next summarise the different OOHDM activities; detailed syntax and semantics can be found in [3,6]. Further information about OOHDM can be found online at the OOHDM Wiki (http://www.ooohdm.inf.puc-rio.br:8668).

10.1.1 Requirements Gathering

The first step during requirements gathering is to gather stakeholders' requirements. To achieve this, it is necessary to first identify the actors (stakeholders) and the tasks they must perform. Next, scenarios are collected (or

drafted), for each task and type of actor. The scenarios are then used to form use cases, which are represented using User Interaction Diagrams (UIDs). These diagrams provide a concise graphical representation of the interaction between the user and the system during the execution of a task. UIDs are validated with the actors, and redesigned if necessary. In sequence, a set of guidelines are applied to the UIDs to extract a conceptual model. Details about UIDs can be found in [9].

10.1.2 Conceptual Design

During the conceptual design, an application domain's conceptual model is built using object-oriented modelling principles, augmented with primitives, such as attribute perspectives (multiple valued attributes, similar to HDM perspectives). Conceptual classes may be built using aggregation and generalisation/specialisation hierarchies. There is no concern for the types of users and tasks, only for the application domain semantics. A conceptual schema is built out of sub-systems, classes and relationships. OOHDM uses UML (with slight extensions) for expressing the conceptual design.

10.1.3 Navigational Design

In OOHDM, an application is seen as a navigational view over the conceptual model. This reflects a major innovation of OOHDM, which recognises that the objects (items) the user navigates are *not* the conceptual objects, but objects that are "built" from one or more conceptual objects.

For each user profile we can define a different navigational structure, which will reflect objects and relationships in the conceptual schema according to the tasks a user must perform. The navigational class structure of a Web application is defined by a schema containing navigational classes. In OOHDM, there is a set of pre-defined types of navigational classes: nodes, links, anchors and access structures. The semantics of nodes, links and anchors are as usual in hypermedia applications. Nodes in OOHDM represent logical "windows" (or views) on conceptual classes, defined during conceptual design. Links are the hypermedia realisation of conceptual relationships, as well as task-related links. Access structures, such as indexes, represent possible ways to start a navigation.

Different applications (in the same domain) may contain different linking topologies according to a user's profile. For example, in an academic Web application we may have a view to be used by students and researchers, and another view for use by administrators. In the second view, a professor's

node may contain salary information, which would not be visible in the student's view.

The main difference between our approach and others', in relation to object viewing mechanisms, is that while others consider Web pages mainly as user interfaces built by "observing" conceptual objects, we favour the explicit representation of navigational objects (nodes and links) during design.

The navigational structure of a Web application is described in terms of navigational contexts, which are generated from navigation classes, such as nodes, links, indices and guided tours. Navigational contexts are sets of related nodes that possess similar navigation alternatives (options), and that are meaningful for a certain step in a task pursued by a user. For example, we can model the set of courses in a semester, the paintings of a painter, the products in a shopping cart, etc.

10.1.4 Abstract Interface Design

The abstract interface design defines perceptible objects (e.g. a picture, a city map) in terms of interface classes. Interface classes are aggregations of primitive classes (e.g. text fields, buttons) and, recursively, of other interface classes. Interface objects are mapped to navigational objects in order to have a perceptible appearance. An interface behaviour is defined by specifying how to handle external and user-generated events, and how the communication between interface and navigational objects is to take place.

10.1.5 Implementation

Implementation maps interface and navigation objects to implementation objects, and may involve elaborated architectures (e.g. client–server), in which applications are clients to a shared database server containing conceptual objects. A number of CD-ROM-based applications, as well as Web applications, have been developed using OOHDM, and employing numerous technologies, such as Java (J2EE), .NET (aspx), Windows (asp), Lua (CGILua), ColdFusion and Ruby (RubyOnRails).

An open source environment for OOHDM, based on a variation of Ruby on Rails, is available at:

http://server2.tecweb.inf.puc-rio.br:8000/projects/hyperde/trac.cgi/wiki.

10.2 Building an Online CD Store with OOHDM

We next illustrate our method using as a case study the design of a simple CD store. To keep it simple, we focus mainly on the process of finding products in the store catalogue, with less emphasis on the check-out process (see [5]). This example is somewhat archetypical, as different Web applications can be modelled using similar ideas to those we show next. We emphasise the process of mapping requirements into conceptual and navigational structures, and ignore user interface and implementation issues (see [1,2,8] for discussions about interfaces and implementation).

In OOHDM we build a different navigational model for each user profile. In this application we have at least two orthogonal profiles: the client (who is looking for CDs to buy) and the administrator (who maintains the CD store); we will illustrate the application focusing on the client profile.

10.2.1 Requirements Gathering

The first step is to identify the actors in the application; in the example, our only actor is the client who buys CDs in the online store. Next, for each actor, we have to identify the tasks that will evolve into potential use scenarios, and later into use cases. The most important tasks identified are the following:

- To buy a CD given its title
- To buy a CD given the name of a song
- To buy a CD given the name of the performer
- To find information about a performer
- To find CDs given a musical genre
- To find best-selling CDs
- To find CDs on offer

Scenario Construction

The next activity consists of describing usage scenarios. Scenarios represent the set of tasks a user has to perform to complete a task. Scenarios in OOHDM are specified textually, from the point of view of the end users. In this instance, the role of an end user (client) can also be performed either by different members of the design team, or by the CD store employees. For the sake of conciseness, we describe two of the eighteen scenarios we elicit from three different users.

Scenario 1: To buy a specific CD.
"I enter the CD title. For each CD matching that title I obtain the CD's cover, availability and price. It is possible to obtain detailed information, such as track names, duration, details of performing artists, and to listen to CD tracks. It is also possible to obtain additional data on artists. After reading the information I decide to buy the CD or to quit"

Scenario 2: To buy a CD given its title.
"I enter the CD title and I obtain the list of matching titles. I choose one and add it to the shopping cart. Whenever the CD information is shown, I should see information on its availability"

Use Case Specification

Next, we define use cases, based on the set of scenarios and tasks previously defined; we use the following heuristics:

1. Identify those scenarios related to the task at hand. We will use the two previous scenarios.
2. For each scenario, identify information items that are exchanged between the user and the application during their interaction.
3. For each scenario, identify data items that are inter-related. In general, they appear together in a use case text.
4. For each scenario, identify data items organised as sets. In general, they appear as sets in a use case text.
5. The sequences of actions presented in scenarios should also be present in a use case.
6. For each scenario, the operations on data items should be included in a use case.

Once the data involved in the interaction, the sequence of actions and the operations have been defined, we next specify a use case. A use case is constructed from the sequence of actions, enriched with data items and operations. Use cases can also be complemented with information from other use cases, or from the designer.

The resulting use case for the previous scenario is the following:

Use Case: To buy a CD from its title

1. A user enters the CD title (or part of it).
2. The application returns a list of matching CDs. If only one CD matches, see step 4. For each CD, its title, artist, price, cover and availability are shown.

3. If the user wants to buy one or more CDs from the list, (s)he adds them to the shopping cart. The sale is dealt with using another use case – *Use Case: Buy*. Further CD information is available by selecting it.
4. If a single CD is selected, the application provides further information: title, cover, availability, price, track names and durations, performers, description, year, genre and country of origin. If the user wants to buy this CD, (s)he can either add it to the shopping cart, or leave and buy it later (*Use Case: Buy*). The user can listen to a track segment if willing to.
5. Further information about any artists who participated in the CD can be obtained by selecting the artist's name. Once selected, the application returns the artist's name, date of birth, a photograph and a short biography.

The specification of the remaining use cases follows a similar process. Thus, only those use cases that are clearly different from the one described above will be described next.

Use Case: Verify Shopping Cart

1. The shopping cart displays information on all the CDs selected by a user. For each CD the following information is provided: title, quantity, artist's name and price. Total price and the estimated delivery date are also shown.
2. The quantity relative to each CD can be edited, if necessary, by selecting the CD.

Use Case: Buy CD

1. To buy CD(s) a user must provide a name and, optionally, a password.
2. If a user does not have a password, the following information must then be provided: name, address, telephone, e-mail address and birth date.
3. Once the necessary information is given, a user is able to further supply the necessary payment data: payment options (cash or deferred), payment type (cheque or credit card), delivery options (surface or air) and optionally delivery address.[1] The operation is completed only after being confirmed by the user.
4. After the operation is confirmed, the user receives an order number.

---

[1] The delivery address only needs to be provided if it differs from the user's contact address.

Specifying User Interaction Diagrams

For each previously defined use case, a User Interaction Diagram (UID) must be specified. The specification of UIDs from use cases can be done following the guidelines described below. As an example, we detail below the process of building the UID for the use case: *To buy a CD given its title.*

1. Initially the use case is analysed to identify the information exchange between the user and the application. Information provided by the user and information returned by the application are tagged. Next, the same information is identified and made evident in the use case.
2. Items that are exchanged during the interaction are shown as the UID's states. Information provided by the user and by the system are always in separate states. Information produced from computations and information used as input to the computations should be in separate states. The ordering of states depends on the dependencies between the data provided by the user, and those returned by the application. In Fig. 10.1, we show the first draft of a UID where parts of the use case are transcribed.

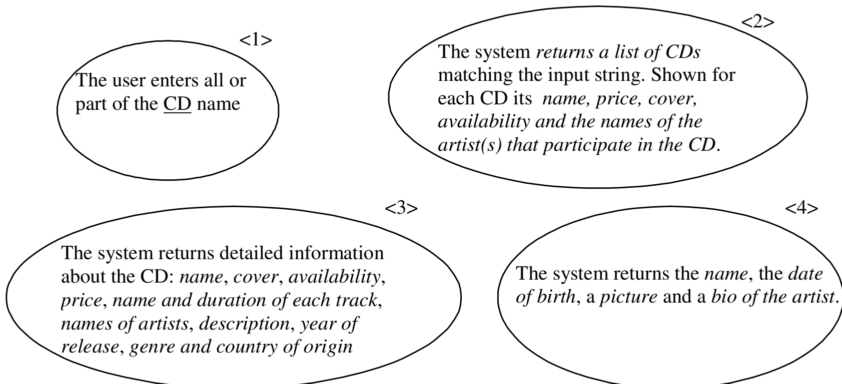

Fig. 10.1. Defining a UID

The exchange data items, once identified, must be clearly indicated in the UID. Data entered by the user (e.g. a CD title) is specified using a rectangle: if it is mandatory, the border is a full line; if it is optional, the border is a dashed line (see Fig. 10.2). An ellipsis (…) in front of a label indicates a list (e.g. …CD indicates a list of CDs). The notation Artist(name, date of birth, bio, photo) is called a *structure*.

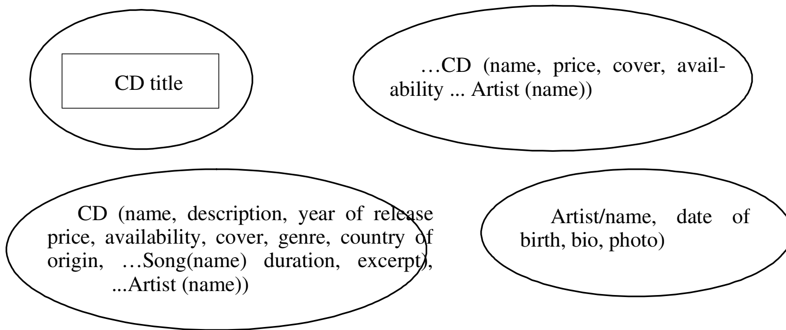

Fig. 10.2. Refining interaction states

Transitions between interaction states must be indicated using arrows. Multiple paths, as indicated in the use cases, might arise (see Fig. 10.3). Labels between brackets indicate conditions (e.g. [2..N] indicates more than one result); a label indicating cardinality represents a choice (in the example, “1” indicates that only one may be chosen).

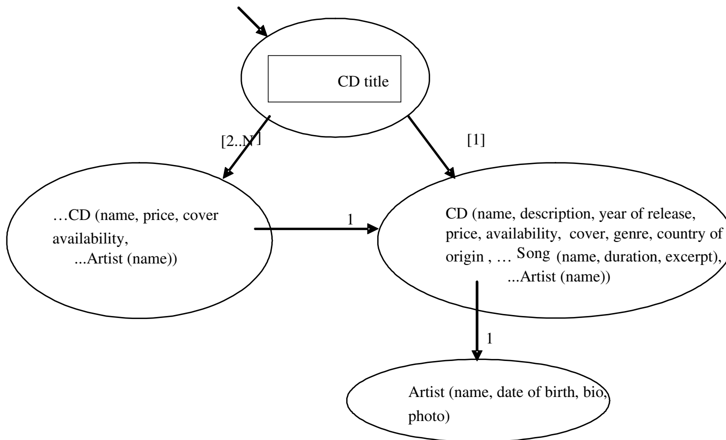

Fig. 10.3. Transitions between interaction states

Finally, operations executed by the user are represented using a line with a bullet connected to the specific information item to which it is applied, as shown in Fig. 10.4. The name of the operation appears in parentheses.

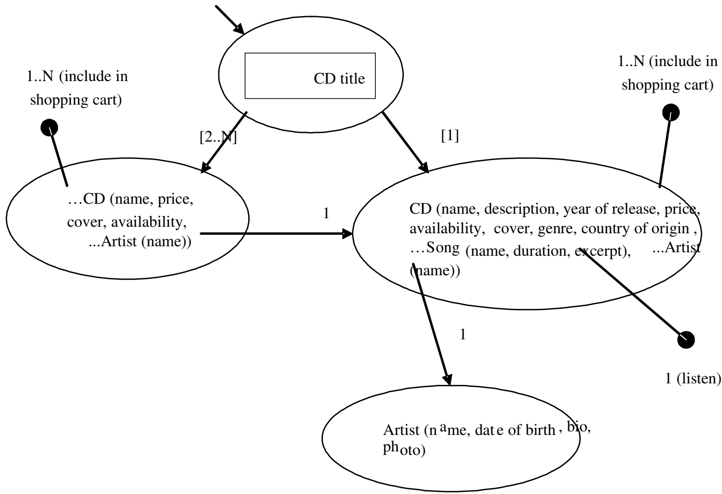

Fig. 10.4. Complete specification of the UID for use case *To buy CD given its title*

Figure 10.5 and Fig. 10.6 present UIDs corresponding to the use cases *To verify Shopping Cart* and *to buy CD*, respectively. Once we finish the specification of UIDs for all use cases, we can then design the application's conceptual model.

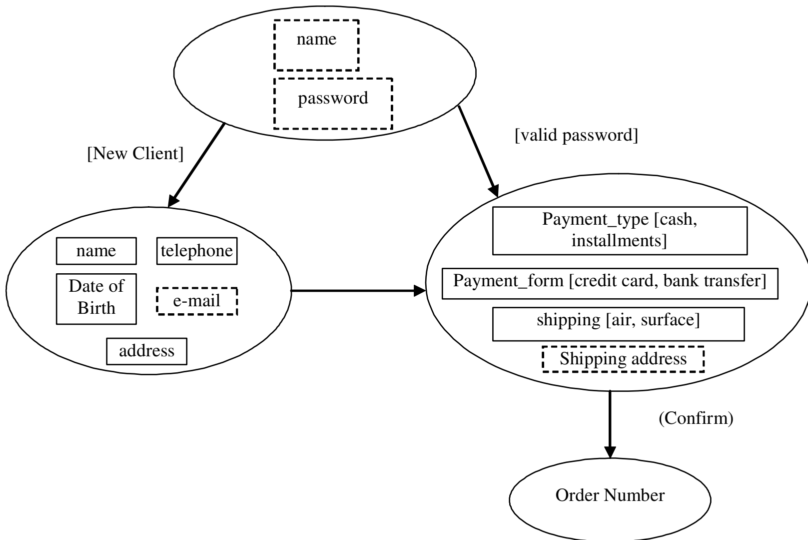

Fig. 10.5. UID for use case *To buy CD*

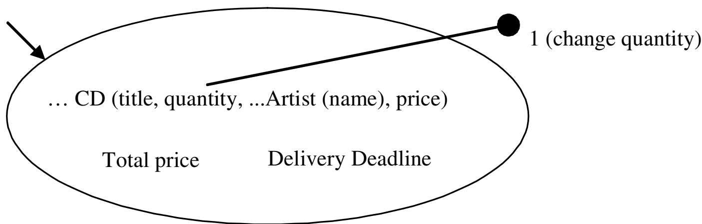

Fig. 10.6. UID for use case *To verify Shopping Cart*

10.2.2 Conceptual Modelling

To define classes, their attributes, operations and relationships is not an easy task. However, the information gathered from use cases and UIDs can help identify core information classes that can be later refined. Next, we describe a set of guidelines to derive classes from UIDs, exemplified using the UID in Fig. 10.4 (To buy CD).

1. **Class definition.** For each data structure in the UID we define a class. In the example, classes are: CD, Artist, Song.
2. **Attribute definition.** For each information item appearing in the UID, either provided by the user or returned by the application, an attribute is defined according to the following validations:

   a. If, given an instance of class *X*, it is possible to obtain the value for attribute *A*, then *A* can be an attribute of *X* (provided *X* is the only class fulfilling this condition).
   b. If, given classes *X* and *Y*, it is possible to obtain the value of attribute *A*, then *A* will be an attribute of an association between *X* and *Y*.
   c. If the attribute corresponding to a data item does not depend on any existing class, or combination of classes, we need to create a new one.

The following attributes were identified from the information returned by the application, as shown in the UID in Fig. 10.4:

- CD: title, description, year, price, cover, availability, genre, country of origin.
- Artist: name, birth date, description, photograph
- Song: name
- CD-Song: track, duration.

3. **Definition of associations.** For each UID, for attributes contained within a structure that does not correspond to their class, include the association if there is a relationship between its class and the class representing the structure.
4. **Definition of associations.** For each UID, for each structure *s1*, containing another structure *s2*, create an association between the classes corresponding to structures *s1* and *s2*.
5. **Definition of associations.** For each transition of interaction states in each UID, if there are different classes representing the source interaction state and the target interaction state, define an association between corresponding classes.

The following associations were identified by applying 3, 4 and 5 to the UID in Fig. 10.4:

- CD-Artist
- CD-Song

6. **Operations definition.** For each option attached to a state transition in each UID, verify if there is an operation that must be created for any of the classes that correspond to the interaction states.

The following operations were identified from this last guideline:

- CD: includeInShoppingCart
- CD-Music: listenTrack

In Fig. 10.7 we show an initial conceptual model derived from the UID: *To buy CD from title*.

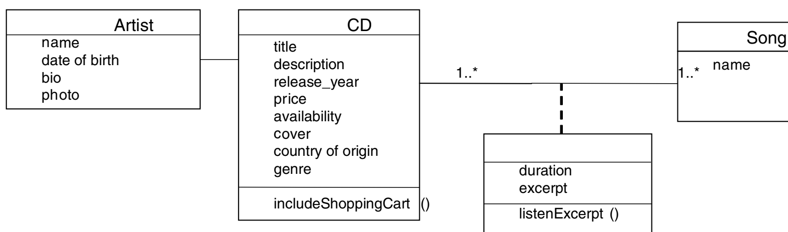

Fig. 10.7. Initial conceptual model

After analysing the complete set of UIDs and performing the required adjustments we obtain the conceptual model shown in Fig. 10.8.

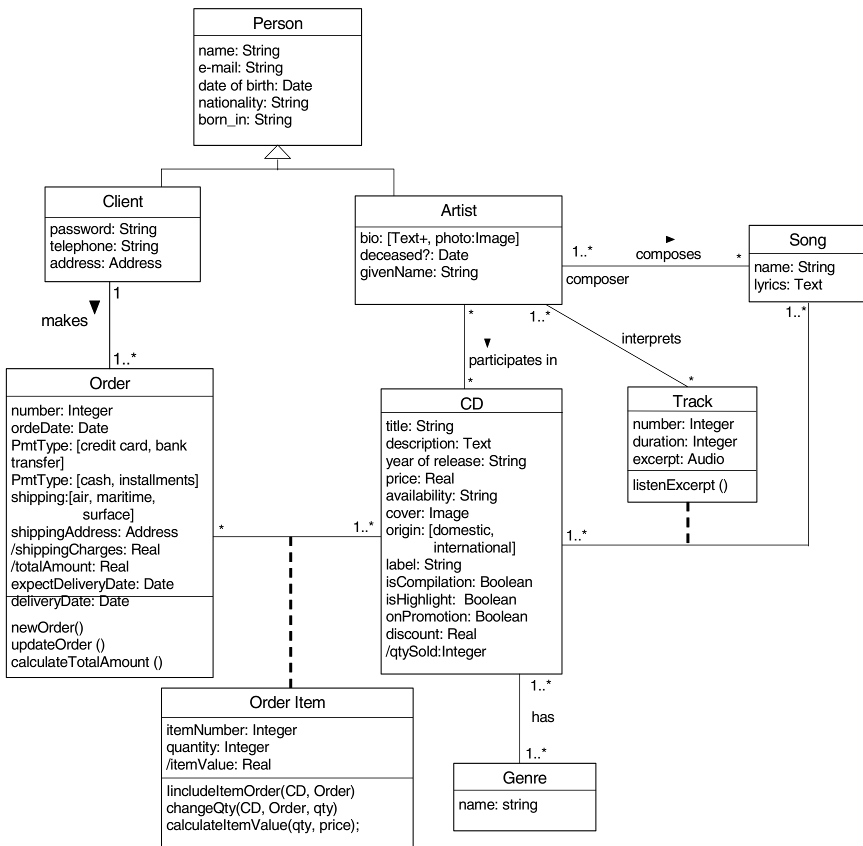

Fig. 10.8. Conceptual model for the CD store

Note that this conceptual model might need further improvements as the application evolves, since these classes are simply the ones we derive from the requirement's gathering activity. However, this evolution belongs more to the general field of object-oriented design and is not important in the context of this chapter.

10.2.3 Navigation Design

During the navigation design activity we generate two schemas: the navigational contexts and the navigational class schemas. The former indicate possible navigation sequences to help users complete their tasks; the latter specify the navigation objects being processed. Designers create both schemas from different sources. UIDs and scenarios are important to obtain a sound navigational model. The conceptual model that has also been

obtained from requirements is also an important source of information. Finally, designers use previous experience, e.g. using navigation patterns, as described in [4,7]. Next we detail the creation of navigational contexts.

Derivation of Navigational Contexts

For each task we define a partial navigational context representing a possible navigational structure to support the task. We detail the creation of the navigational contexts corresponding to the use case: *To buy CD given its title.*

First, each structure that has been represented in the UID (and the corresponding class in the conceptual model) is analysed to determine the type of primitive that it will give rise to (e.g. an access structure, a navigational context or a list). The following guidelines can be used to obtain a navigational context:

1. When the task associated with the UID requires that the user examines a set of elements to select one, we map the set of structures into an access structure. An access structure is a set of elements, each of which contains a link. In Fig. 10.9, we show the partial diagram for access structures CDs and Artists.

Fig. 10.9. Access structures

2. When the task does not require such examination, but requires the elements to be accessed simultaneously, map the set into a list, e.g. the list of songs in a CD (see Fig. 10.10).

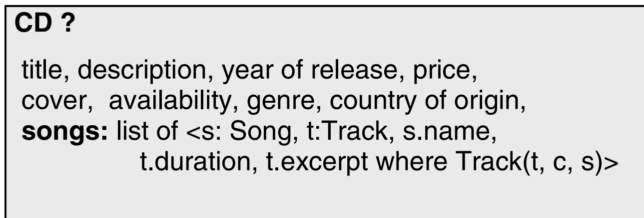

Fig. 10.10. List for CD

3. After mapping the different sets of structures we analyse singular structures in the UID using the following guideline. When the task requires that an element's information be accessed by the user, we map the structure into a navigational context. In Fig. 10.11 we show the partial context diagram from this example.

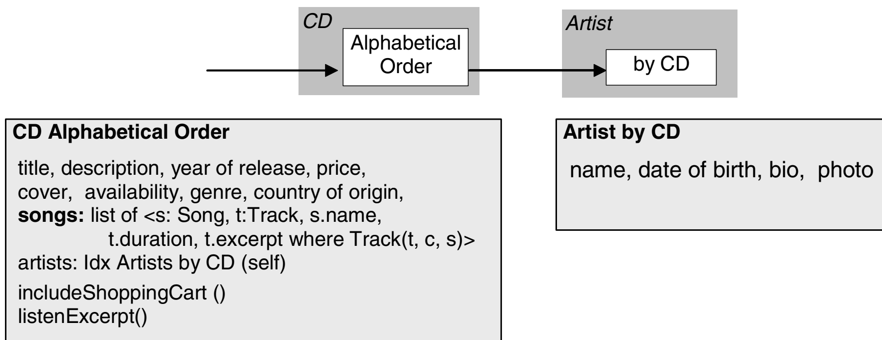

Fig. 10.11. Partial context for UID: *Buy CD given its title*

In the example, both "CD Alphabetical Order" and "Artist by CD" are contexts, which correspond to sets of elements. The elements that constitute each set are described in the grey boxes.

In Fig. 10.12 and Fig. 10.13 we show other partial contexts obtained from previously mentioned UIDs. Other UIDs, such as "CD by Genre", "CD on Promotion", would have similar definitions.

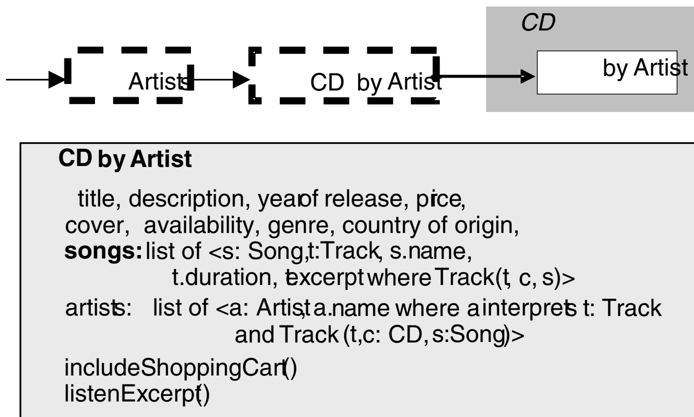

Fig. 10.12. Partial context for UID: *To buy CD given an artist's name*

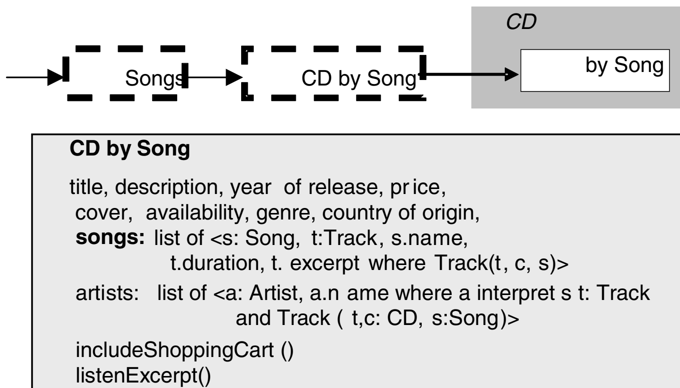

Fig. 10.13. Partial context for UID: *To buy CD given a song's name*

Fig. 10.14 and Fig. 10.15 show other kinds of contexts and their element definitions.

After obtaining the context diagram for each individual task, we integrate the partial context schemas to obtain the application's complete navigational context schema, shown in Fig. 10.16. In the integration process, contexts that are the same are unified, and navigation choices between contexts in different tasks are also examined.

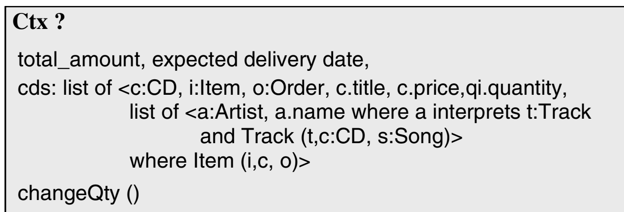

Fig. 10.14. Verify shopping cart

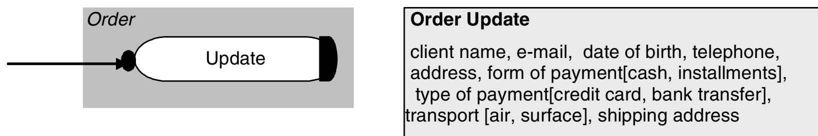

Fig. 10.15. To buy CD

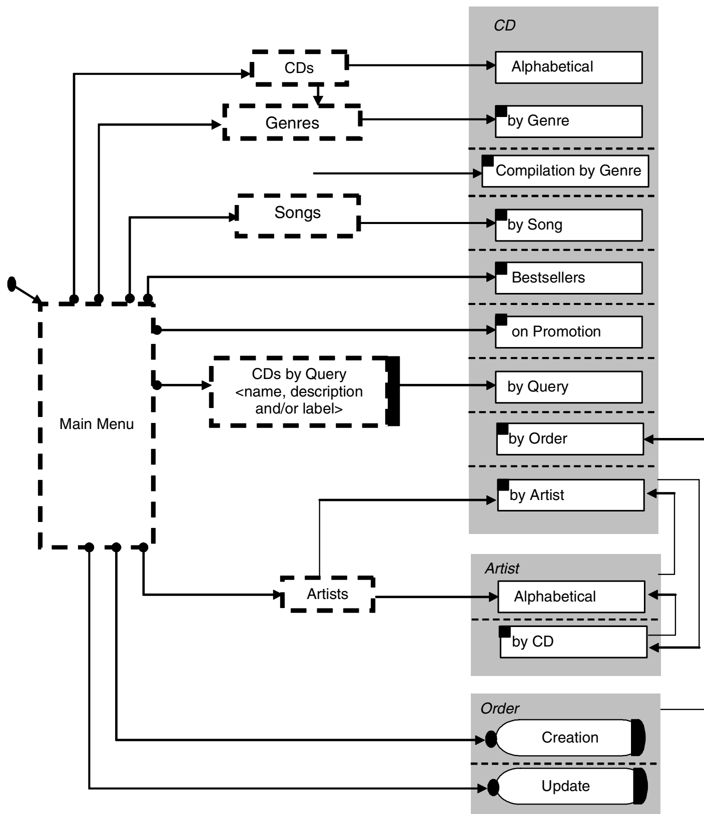

Fig. 10.16. Navigation context schema

We can see that from the main menu, the user can access different access structures (for CDs, Musical Genres, Songs, CDs by Query, and Artists). Each one of them provides access to sets of nodes that support the achievement of the different tasks identified at the outset.

Specification of the Navigational Class Schema

During the specification of the navigational class schema the designer derives the navigational schema using both the conceptual model and the navigational contexts schema. Navigational classes, such as nodes, represent views over conceptual classes: a navigational class can present information from one or more conceptual classes. All classes from the navigational contexts schema are node classes. Meanwhile links are derived from navigation relationships between classes in the navigational contexts schema. Note that not all navigation in this schema represents a link. The rule for selecting the target context is analysed (especially when it involves

navigation between contexts of the same class). If the elements of the target context are related to an object of the same original class, and if this object is the parameter, then the navigation represents a link.

For example, in the navigational context schema of Fig. 10.16, we have navigational classes *CD*, *Order* and *Artist*. We have the following navigations among classes: from *CD* to *Artist by CD*, from *Order* to *CD by Order* and from *CD* to *Order in Creation/Update*. The selection rule for *Artist by CD* (Parameter: *c:CD-Elements: a: Artist* where *a participates in c*) indicates that the context is integrated by artists related to a particular CD, which is its parameter; therefore there is a link from *CD* to *Artist*. Similarly, selection rules for the other contexts indicate which navigations correspond to links. In Fig. 10.17 we present the resulting navigational class schema.

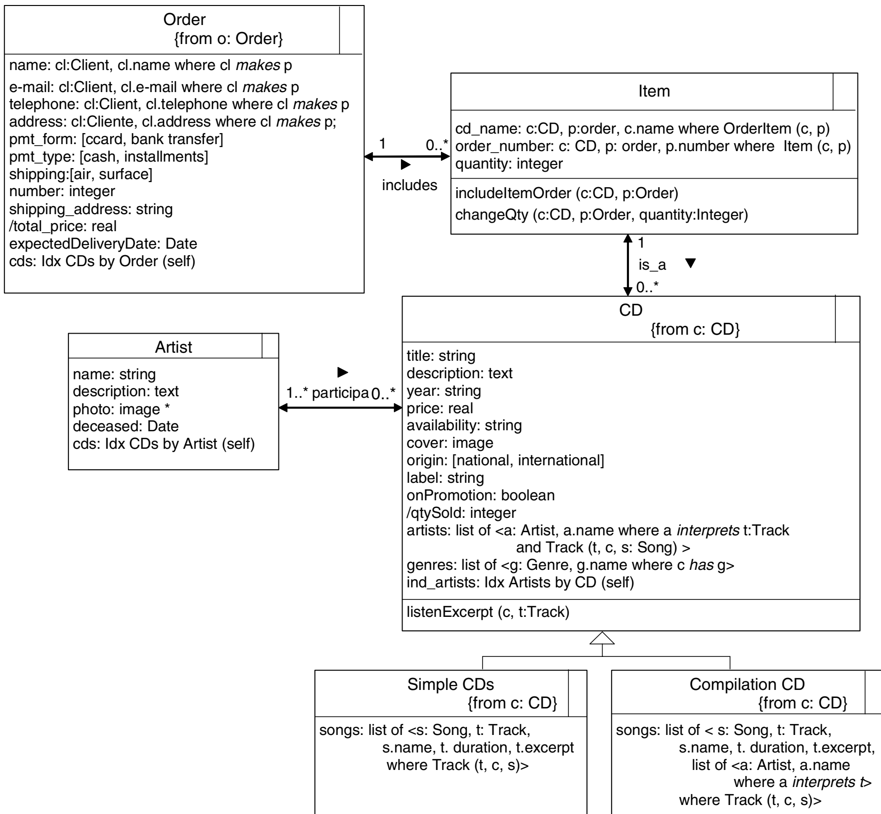

Fig. 10.17. Navigational schema

10.2.4 Abstract Interface Design

The abstract interface design focuses on making navigation objects and application functionality perceptible to the user, which must be done at the application interface level. At the most abstract level, the interface functionality can be regarded as supporting information exchange between the application and the user, including activation of functionalities. In fact, from this standpoint, navigation is just another (albeit distinguished) application functionality.

Since the tasks being supported drive this information exchange, it is reasonable to expect that this exchange in itself will be less sensitive to runtime environment aspects, such as particular standards and devices being used. The design of this interface aspect can be carried out by interaction designers or software engineers.

For the actual running application, it is necessary to define the concrete look and feel of the application, including layout, font, colour and graphical appearance, which is typically carried out by graphics designers. This part of the design is almost totally dependent on the particular hardware and software runtime environment.

Such separation allows shielding a significant part of the interaction design from inevitable technological platform evolution, as well as from the need to support users in a multitude of hardware and software runtime environments.

The entire interface is specified by several ontologies, currently described using RDFS (RDFS W3C) and OWL (OWL W3C) as a formalism.

Abstract Widget Ontology

The type of functionality offered by interface elements is called the abstract interface. It is specified using the Abstract Widget Ontology, which establishes the interface vocabulary, as shown in Fig. 10.18. This ontology can be thought of as a set of classes whose instances will comprise a given interface.

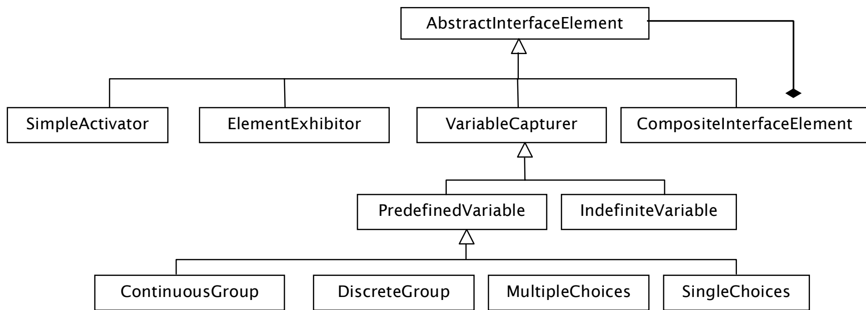

Fig. 10.18. Abstract Widget Ontology

An abstract interface widget can be any of the following:

- *SimpleActivator,* which is capable of reacting to external events, such as mouse clicks.
- *ElementExhibitor,* which is able to exhibit a type of content, such as text or images.
- *VariableCapturer,* which is able to receive (capture) the value of one or more variables. This includes input text fields, selection widgets such as pull-down menus and checkboxes, etc. It generalises two distinct (sub) concepts.
- *IndefiniteVariable,* which allows entering previously unknown values, such as text strings typed by the user.
- *PredefinedVariable,* which abstracts widgets that allow the selection of a subset from a set of pre-defined values; often this selection must be a singleton. Specialisations of this concept are *ContinuousGroup*, *DiscreetGroup*, *MultipleChoices* and *SingleChoice*. The first allows selecting a single value from an infinite range of values; the second is analogous, but for a finite set; the remainder are self-evident.
- *CompositeInterfaceElement,* which is a composition of any of the above.

It can be seen that this ontology captures the essential roles that interface elements play with respect to the interaction – they exhibit information, react to external events, or accept information. As customary, composite elements allow building more complex interfaces out of simpler building blocks.

The software designer, who understands the application logic and the types of information exchange that must be supported, should carry out the abstract interface design. The software designer does not need to take usability issues or the "look and feel" into account, as they will be dealt with during the concrete interface design, normally carried out by a graphics (or "experience") designer.

Once the abstract interface has been defined, each element must be mapped onto both a navigation element, which will provide its contents, and a concrete interface widget, which will actually implement the element in a given runtime environment. Fig. 10.19 provides an example of an interface for a page describing an artist, and Fig. 10.20 shows an abstract representation of this interface.

Concrete widgets correspond to widgets usually available in most runtime environments, such as labels, text boxes, combo boxes, pulldown menus, radio buttons, etc.

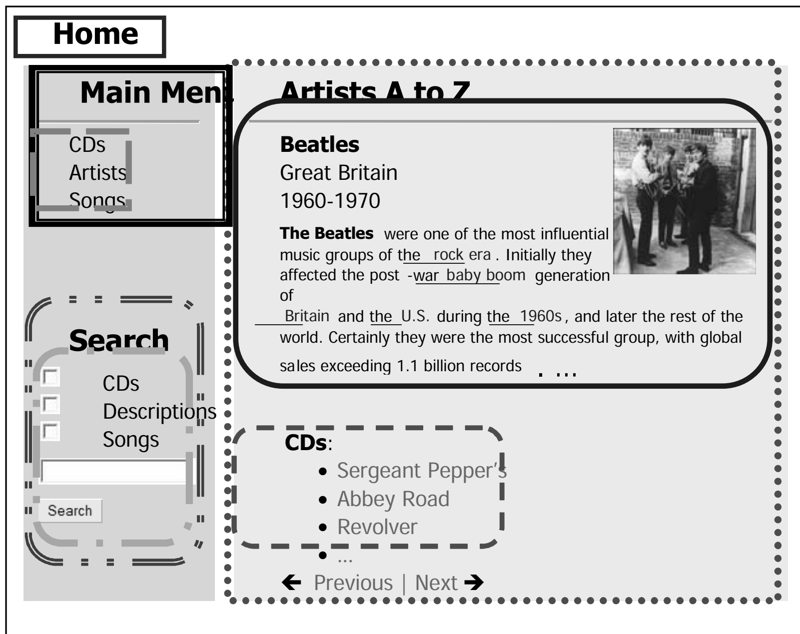

Fig. 10.19. An example of a concrete interface

Fig. 10.20. Abstract Widget Ontology instance for the example in Fig. 10.19

Mappings

The Abstract Interface Ontology contains, for each abstract interface widget, the mapping both to navigation elements, which are application specific, and to a concrete interface element.

There is additional information in the ontology that restricts each abstract interface widget to compatible concrete interface widgets. For example, a "SimpleActivator" abstract interface widget can only be mapped into the "Link" or "Button" concrete interface widgets.

Actual abstract interface widget instances are mapped onto specific navigation elements (in the navigation ontology) and onto concrete interface widgets (in the Concrete Interface Widget Ontology). Fig. 10.21 shows the specification of the "Previous Artist" abstract interface widget (class "SimpleActivator"), shown in Fig. 10.20, which is mapped onto a "Link" concrete interface element.

```
...
<awo:SimpleActivator rdf:ID="ArtistAlphaPrevious">
        <awo:mapsTo rdf:resource= "http://www.inf.puc-rio.br/~sabrina/ontology/CW/cwo#Link" />
        <awo:fromElement>ctxArtistAlpha</awo:fromElement>
        <awo:fromAttribute>_Prev</fromAttribute>
        <awo:AbstractInterface>ArtistAlpha</AbstractInterface>
</awo:SimpleActivator>
```

Fig. 10.21. Mapping between abstract interface widget and navigation element

Fig. 10.22 shows an example illustrating how an application's functionality is integrated, by providing the OWL specification of the "Search" abstract interface element. It is composed of two abstract widgets, "ElementExhibitor" (lines 9–12), and "CompositeInterfaceElement" (lines 14–46). The first shows the "Search" string, using a "Label" concrete widget. The second aggregates the four elements used to specify the field in which the search may be performed, namely, three "MultipleChoices" – SearchProfessors (lines 25–29), SearchStudents (31–35) and SearchPapers (37–41) and one "IndefiniteVariable" – "SearchField" (lines 43–46).

The *CompositeInterfaceElement* element, in this case, has the properties: *fromIndex*, *isRepeated*, *mapsTo*, *abstractInterface* and *hasInterfaceElement*. The *fromIndex* property in line 2 indicates which navigational index this element belongs to. This property is mandatory if no previous element of type *compositeInterfaceElement* has been declared. The association with the "idxSearch" navigation element in line 2 enables the generation of the link to the actual code that will run the search. Even though this example shows an association with a navigation element, it could just as well be associated with a call to application functionality such as "buy".

```
...
1   <awo:CompositeInterfaceElement rdf:ID="Search">
2       <awo:fromIndex>idxSearch</awo:fromIndex>
3       <awo:mapsTo rdf:resource="&cwo;Composition"/>
4       <awo:isRepeated>false</awo:isRepeated>
5       <awo:hasInterfaceElement rdf:resource="#TitleSearch"/>
6       <awo:hasInterfaceElement rdf:resource="#SearchElements"/>
7   </awo:CompositeInterfaceElement>
8
9   <awo:ElementExihibitor rdf:ID="TitleSearch">
10      <awo:visualizationText>Search</awo:visualizationText>
11      <awo:mapsTo rdf:resource="&cwo;Label"/>
12  </awo:ElementExihibitor>
13
14  <awo:CompositeInterfaceElement rdf:ID="SearchElements">
15      <awo:fromIndex>idxSearch</awo:fromIndex>
16      <awo:abstractInterface>SearchResult</awo:abstractInterface>
17      <awo:mapsTo rdf:resource="&cwo;Form"/>
18      <awo:isRepeated>false</awo:isRepeated>
19      <awo:hasInterfaceElement rdf:resource="#SearchCDs"/>
20      <awo:hasInterfaceElement rdf:resource="#SearchDescriptions"/>
21      <awo:hasInterfaceElement rdf:resource="#SearchSongs"/>
22      <awo:hasInterfaceElement rdf:resource="#SearchField"/>
23  </awo:CompositeInterfaceElement>
24
25  <awo:MultipleChoices rdf:ID="SearchCDs">
26      <awo:fromElement>SearchCDs</awo:fromElement>
27      <awo:fromAttribute>name</awo:fromAttribute>
28      <awo:mapsTo rdf:resource="&cwo;CheckBox"/>
29  </awo:MultipleChoices>
30
31 <awo:MultipleChoices rdf:ID="SearchDescriptions">
32      <awo:fromElement>SearchCDs</awo:fromElement>
33       <awo:fromAttribute>description</awo:fromAttribute>
34      <awo:mapsTo rdf:resource="&cwo;CheckBox"/>
35  </awo:MultipleChoices>
36
37  <awo:MultipleChoices rdf:ID="SearchSongs">
38      <awo:fromElement>SearchSongs</awo:fromElement>
39      <awo:fromAttribute>name</awo:fromAttribute>
40      <awo:mapsTo rdf:resource="&cwo;CheckBox"/>
41  </awo:MultipleChoices>
42
43  <awo:IndefiniteVariable rdf:ID="SearchField">
44      <awo:mapsTo rdf:resource="&cwo;TextBox"/>
4546  </awo:IndefiniteVariable>
...
```

Fig. 10.22. Example of the OWL specification of the "Search" part of Fig. 10.19

The *isRepeated* property indicates if the components of this element are repetitions of a single type (false in this case). The *mapsTo* property indicates which concrete element corresponds to this abstract interface element. The *abstractInterface* property specifies the abstract interface that will be activated when this element is triggered. The *hasInterfaceElement* indicates which elements belong to this element.

The *ElementExhibitor* element has the *visualizationText* and *mapsTo* properties. The former represents the concrete object to be exhibited, in this case the string "Search".

The *MultipleChoices* element has the *fromElement*, *fromAttribute* and *mapsTo* properties. The *fromElement* and *fromAttribute* properties indicate the corresponding element and navigational attribute in the navigational ontology, respectively. The *IndefiniteVariable* element has the *mapsTo* property.

10.3 From Design to Implementation

Mapping design documents into implementation artefacts is usually time-consuming and, in spite of the importance of software engineering approaches be generally accepted, implementers tend to overlook the advantages of good modelling practices. The relationship between design models and implementation components is lost, making the traceability of design decisions, which is a fundamental aspect for supporting evolution, a nightmare. We claim that this problem is not only caused by the relative youth of Web implementation tools but mainly due to:

- Lack of understanding that navigation (hypertext) design is a defining characteristic of Web applications.
- The fact that languages and tools are targeted more to support fine-grained programming than architectural design.
- The inability of methodologists to provide non-proprietary solutions to the aforementioned “mapping” dilemma.

For example, we can use the Model View Controller (MVC) architecture to map design constructs onto implementation components. The MVC architecture has been extensively used for decoupling the user interface from application data, and from its functionality. Different programming environments provide large class libraries that allow the programmer to reuse standard widgets and interaction styles by plugging corresponding classes into her/his “model”.

The model contains application data and behaviours, and also provides an interface for the view and the controller. For each user interface, a view object is defined, containing information about presentation formats, and is kept synchronised with the model’s state. Finally, the controller processes the user input and translates it into requests for specific application’s functionality. This separation reflects well the fact that Web applications may have different views, in the sense that it can be accessed through different clients (e.g. browsers, WAP clients, Web service clients), with application data separated from its presentation. The existence of a separate module (the controller) to handle user interaction, or, more generally, interaction

with other systems or users, provides better decoupling between application behaviour and the way in which this behaviour is triggered.

However, while the MVC provides a set of structuring principles for building modular interactive applications, it does not completely fulfil the requirements of Web applications to provide rich hypermedia structures, as it is based on a purely transactional view of software. In addition, it does not take into account the navigation aspects that, as we have previously argued, should be appropriately supported.

The view component includes structure and presentation of data, while contents are kept in the model. Specifically, a simple use of the MVC is for nodes and their interfaces to be handled by the same software component (typically a JSP object).

In addition, the MVC does not take into account that navigation should always occur within a context and that context-related information should be provided to the user. For example, if we want the same node to have a slightly different structure, depending on the context in which it is accessed (e.g. CD in a thematic set or in the shopping cart), we have to use the context as a parameter for the JSP page, and write conditional statements to insert context-sensitive information as appropriate. The JSP becomes overloaded, difficult to manage and evolution becomes practically unmanageable. The same problem occurs if we use different JSPs for different contexts, thus duplicating code.

An alternative approach is to use a single JSP that generates the information common to all contexts (basic node), and one JSP for each node in context, which dynamically inserts that common JSP, adding the context-sensitive information. This is still unsatisfactory, since in this case, the basic node layout becomes fixed and we have lost flexibility.

To overcome these limitations we have developed a software architecture, OOHDM-Java2, which extends the idea of the MVC by clearly separating nodes from their interfaces, thus introducing navigation objects; it also recognises the fact that navigation may be context-dependent. Details on the architecture are presented in [1].

In Fig. 10.23 the higher-level components of the OOHDM-Java2 architecture are presented, in addition to the most important interactions between components, while handling a request.

The main components of OOHDM-Java2 are summarised in Table 10.2.

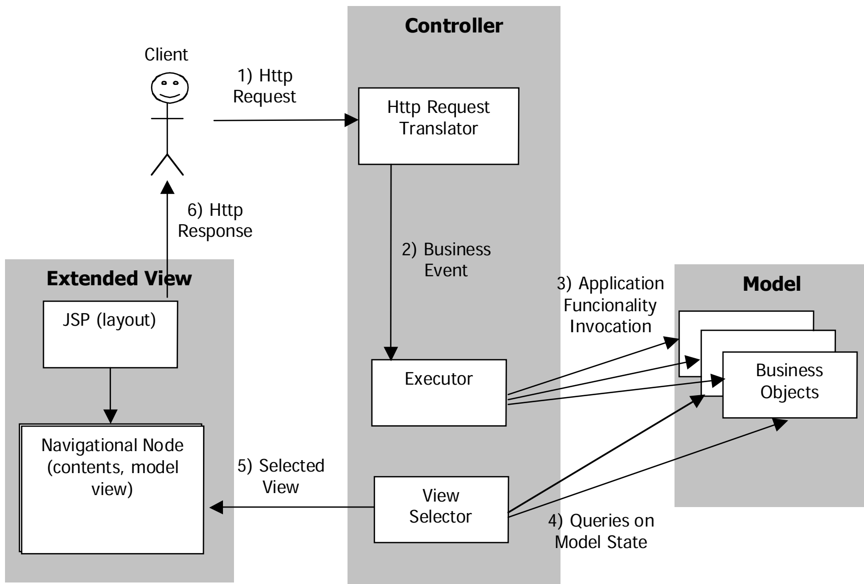

Fig. 10.23. Main components of OOHDM-Java2

Fig. 10.24 outlines the implementation architecture for the interface [2]. Starting with the navigation and abstract interface designs, the corresponding ontology instances are used as input into a JSP generator, which instantiates the interface as a JSP file using TagLibs. The interpreter uses the Jena library to manipulate the ontology information.

The actual TagLib code used is determined by the concrete widget definition that has been mapped onto the corresponding abstract widget. The abstract interface determines the nesting structure of elements in the resulting page. It is expected that the designer will group together functionally-related elements.

It is possible to use different instances of the TagLib implementation by changing its declaration. Thus, for each possible concrete widget, a different implementation of the TagLib code will generate the desired HTML (or any other language) version for that widget.

Table 10.2. Main components of OOHDM-Java2

Component	Description
HTTP Request Translator (Controller)	Every http request is redirected to this component. It translates the user request into an action to be executed by the model. This component extracts the information (parameters) of the request and instantiates a business event, which is an object that encapsulates all data needed to execute the event.
Executor (Controller)	This component has the responsibility of executing a business event, invoking model behaviours following some predefined logic.
Business Object (Model)	This component encapsulates data and functionality specific to the application. All business rules are defined in these objects and triggered from the executor to execute a business event.
View Selector (Controller)	After the execution of a business event, this component gets the state of certain business objects and selects the response view (interface).
Navigational Node (Extended View)	This component represents the product of the navigational logic of the application; it encapsulates attributes that have been obtained from some business objects and other navigational sub-components such as indexes, anchors, etc. This component has the contents to be shown by the response interface (JSP).
JSP (Extended View)	This component generates the look-and-feel that the client component receives as a response to its request. To achieve this, it instantiates the corresponding navigational node component and adds the layout to the node's contents. Notice that the JSP component does not interact directly with model objects. In this way we can have different layouts for the same navigational node.

The actual values of navigation elements manipulated in the page are stored in Java Beans, which correspond to the navigation nodes described earlier. The *element* property, generated in the JSP file, contains calls to the bean that the Tag Library uses to generate the HTML code seen.

Our current implementation of the TagLib code simply wraps each element that has the "DIV" CSS tag with its own ID, and its CSS class is defined according to its abstract widget type. In this way, we can attach CSS style sheets to the generated HTML to produce the final page rendering.

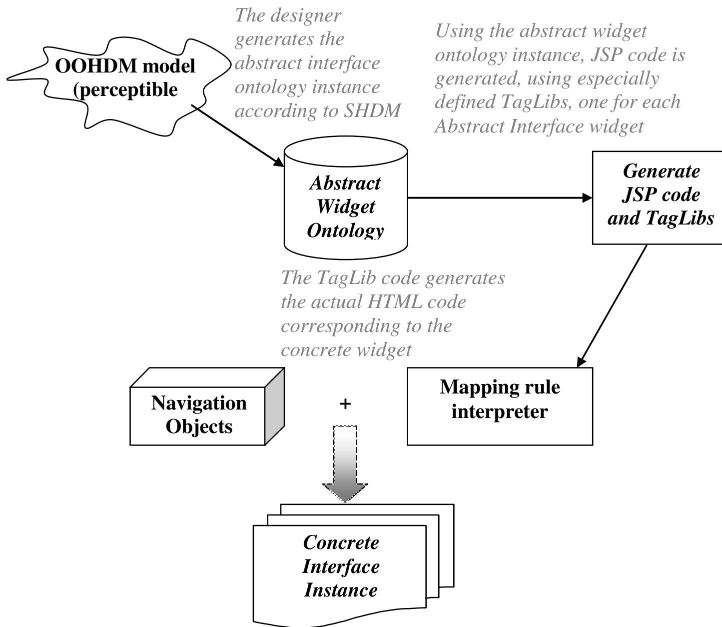

Fig. 10.24. Outline of the implementation architecture

Given the expressive power of CSS, the concrete page definition format allows a large degree of flexibility for the graphic designer, both in terms of layout itself and in terms of formatting aspects. Nevertheless, if a more elaborate page layout is desired, it is possible to edit the generated JSP manually, altering the relative order of generated elements. For a more automated approach, it might be necessary to apply XSLT transformations to the JSP.

10.4 Discussion and Lessons Learned

One of the main advantages of using a model-based approach for Web applications' design is the construction of a set of technology-independent models that can evolve together with application requirements, and that are largely neutral with respect to other types of changes in the application (e.g. runtime settings change).

While working with the OOHDM approach we have found that stakeholders feel comfortable with our notation for requirements acquisition (UID diagrams). In addition, we have used this notation several times to discuss requirements and requirements evolution.

The transition from requirements to design can be managed in a seamless way (perhaps simpler than the transition to implementation). Regarding the implementation, we have found that the instability of frameworks for Web applications deployment usually hinders the use of model-based approaches, as developers tend to devote much time to implementation and to neglect design aspects. In this sense, we have tried to keep our notation simple to make it easy to use.

10.5 Concluding Remarks

This chapter presented the OOHDM approach for building Web applications. We have shown with a simple but archetypical example how to deal with the different activities in the OOHDM life cycle. We have also presented several guidelines that allow a designer to systematically map requirements to conceptual and navigational structures. Finally, implementation alternatives have also been discussed.

Web engineering is no longer in its infancy; many mature methods already exist and developers can base their endeavours on solid model-based approaches like OOHDM and others in this book. The underlying principles behind OOHDM, essentially the clear separation of concerns (e.g. conceptual from navigational and navigational from interfaces), allow not only "just in time" development but also seamless evolution and maintenance of complex Web applications.

Acknowledgements

The authors wish to thank the invaluable help of Adriana Pereira de Medeiros in preparing the example used in this chapter. Gustavo Rossi has been partially funded by Secyt's project PICT No 13623, and Daniel Schwabe has been partially supported by a grant from CNPq - Brazil.

References

1 Jacyntho MD, Schwabe D, Rossi G (2002) A software Architecture for Structuring Complex Web Applications. Web Engineering, 1(1)

2 Moura SS, Schwabe D (2004) Interface Development for Hypermedia Applications in the Semantic Web. In: Proceedings of LA Web 2004, Ribeirão Preto, Brazil, IEEE CS Press, pp 106–113, Los Alamitos, CA

3 Rossi G, Schwabe D (1999) Web application models are more than conceptual models. In: Proceedings of the World Wild Web and Conceptual Modeling'99 Workshop, LNCS 1727, Springer, Paris, pp 239–252

4 Rossi G, Schwabe D, Lyardet F (1999) Integrating Patterns into the Hypermedia Development Process. New Review of Hypermedia and Multimedia, December

5 Schmid H, Rossi G (2004) Modeling and Designing Processes in E-commerce Applications. IEEE Internet Computing, January/February: 19–27

6 Schwabe D, Rossi G (1998) An Object Oriented Approach to Web-Based Application Design. Theory and Practice of Object Systems, 4(4):207–225

7 Schwabe D, Rossi G, Lyardet F (1999) Improving Web Information Systems with navigational patterns. Computer Networks and Applications, May

8 Schwabe D, Szundy G, de Moura SS, Lima F (2004) Design and Implementation of Semantic Web Applications. In: Proceedings of the Workshop on Application Design, Development and Implementation Issues in the Semantic Web (WWW 2004), CEUR Workshop Proceedings, http://ceur-ws.org/Vol-105/, May

9 Vilain P, Schwabe D, Souza CS (2000) A Diagrammatic Tool for Representing User Interaction in UML. In: Proceedings UML'2000, LNCS 1939, Springer Berlin, pp 133–147

Authors' Biography

Gustavo Rossi is full Professor at Facultad de Informática of La Plata National University, Argentina, and heads LIFIA, a computer science research lab. His research interests include Web design patterns and frameworks. He coauthored the Object-Oriented Hypermedia Design Method (OOHDM) and is currently working on separation of design concerns in context-aware Web applications. He has a PhD in Computer Science from Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He is an ACM member and IEEE member.

Daniel Schwabe is an Associate Professor in the Department of Informatics at Catholic University in Rio de Janeiro (PUC), Brazil. He has been working on hypermedia design methods for the last 15 years. He is one of the authors of HDM, the first authoring method for hypermedia, and of OOHDM, one of the mature methods in use by academia and industry for Web applications design. He earned a PhD in Computer Science in 1981 at the University of California, Los Angeles.