

A Catalogue of Refactorings for Navigation Models

Jordi Cabot
Universitat Oberta de Catalunya
Barcelona, Spain
jcabot@uoc.edu

Cristina Gómez
Universitat Politècnica de Catalunya
Barcelona, Spain
cristina@lsi.upc.edu

Abstract

The evolution of web applications (from read-only applications for browsing the data to full-fledged content-modification applications) has increased the complexity of navigation models describing the set of web pages included in a web application.. In this paper, we propose adopting the refactoring technique to reorganize and improve the quality of such models. This technique was initially proposed to improve the structure of source code without changing its external observable behaviour. We adapt the refactoring technique to the navigation models context and present a catalogue of refactorings specific for this particular kind of models.

1. Introduction

Many web development methods are evolving to cover the definition of full-fledged web applications, including data processing and manipulation functionalities. As a consequence, the models involved in the specification of a web application (i.e, the data model to specify the data used by the application, the navigation model to describe the organization of its front-end interface and the presentation model to personalize its graphical aspect) have been extended with new modelling primitives.

The most relevant extensions are the addition of read operations to indicate the data that must be shown in a given page and content-management primitives to denote the modifications to be applied on the database (or in general, any kind of persistent storage) in response to the user actions.

Including these new primitives in web models increase their complexity. This is especially true for navigation models. Even for small web applications, navigation models can become very huge and complex, which jeopardizes their quality. This is a critical issue since the quality of navigation models has a direct effect on the quality and maintainability of the final

web application, usually (semi-) automatically derived from the web model designs.

This problem has been thoroughly studied for object-oriented software systems with large amounts of source code. In this field, the refactoring technique [1] has been successfully proposed to improve the structure and the quality of the code (resulting in a simpler and more readable program). Refactoring is “the process of changing an object-oriented software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure” [2].

We believe that the same idea could help designers to improve the internal structure of navigation models. Therefore, in this paper we propose to adapt the refactoring technique to a web context. For this purpose we need to: 1 – reformulate the concept of behaviour preservation for navigation models and 2 – provide a meaningful catalogue of refactorings specific for this kind of models.

With our proposal, designers will be able to improve and restructure their navigation models with the confidence that the evolved model is behaviour preserving with respect to the initial one. In our proposal, navigation models are formally represented as directed graphs while refactorings are expressed as graph transformation rules.

As far as we know, ours is the first approach to formalize the application of refactorings (or, in general, any set of behaviour preserving transformations) to help in improving the quality of existing navigation models. The notion of refactorings for web applications has also been informally introduced in [3] but there the key concept of behaviour preservation to ensure the applicability of the refactorings is not addressed. Additional related research is focused on the definition of design patterns for navigation models, known as navigation patterns (see [4], [5] as examples). However, these navigation patterns are aimed at guiding the process of manually creating new navigation models from scratch and, in general, are not presented in a formal way but just intuitively

described. Moreover, behavioural issues are not usually included.

The rest of the paper is structured as follows. Section 2 reviews some basic concepts. Section 3 formalizes our graph-based representation for navigation models. Section 4 adapts the concept of behaviour preserving to navigation models and Section 5 presents our catalogue of refactorings. Section 6 presents our tool support. Finally, Section 7 sketches the requirements for an automatic refactoring application and Section 8 presents some conclusions and further work.

2. Basic Concepts of Navigation Models

A navigation model (also known as a hypertext model) specifies the organization of the front-end interface of a web application.

The main elements of navigation models are pages and links. As an example, Fig. 2.1 shows a small excerpt of a possible navigation model (in WebML [6]) for an e-commerce application. In particular, the model shows the interface to create new sales and the related sale lines.

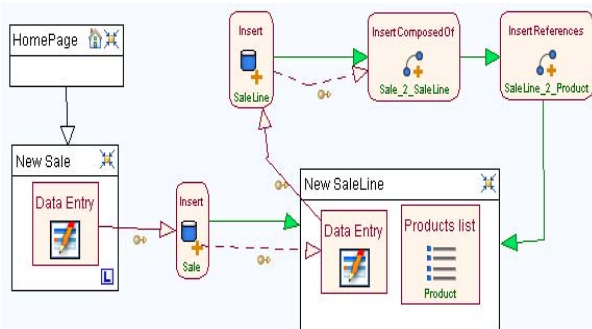


Figure. 2.1. A fragment of a navigational model for an e-commerce application

Pages may include several modelling constructs to specify the page contents. In particular, pages may include *read* operations over the underlying application data. The result of these queries is used to dynamically build the page contents at run-time. This requires the navigation model to be strongly related with a corresponding data model that specifies the information managed by the web application. As an example, a possible data model for the same e-commerce application could be the one shown in Fig 2.2. The main constructs in data models are entity types (i.e. classes), relationship types (i.e. associations) and generalizations. An *entity type ET* describes the common characteristics of a set of entities (i.e. objects) of the domain. Each *ET* contains a set of *attributes*. A binary *relationship type RT* has a name and two participants, each one playing a certain role in the

relationship type. Each relationship (i.e. link) between two participant entities represents a semantic connection between them.

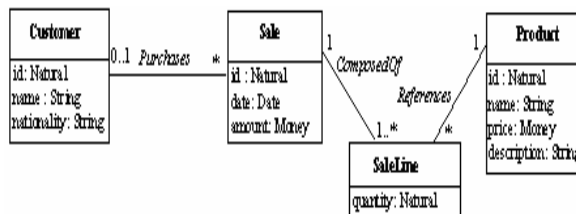


Figure. 2.2. A data model for the e-commerce application

We assume that navigation models may include a single type of read operation: *ReadET*(*att*₁...*att*_n, *selector*) that returns the *att*₁...*att*_n attribute values of the entities of type *ET* satisfying the *selector* condition. For instance, in Fig. 2.1, *NewSaleLine* retrieves the names of available *Products* to help selecting the product to buy. In the figure, this read operation is specified as a WebML index unit with no selector condition (retrieved attributes are not graphically shown in WebML). A page may show information from different entity types by combining several read operations. Note that read operations are attached to pages and not to links since it is during the process of rendering the page when the data that must be shown in the page is computed. However, links may carry parameter values that restrict at run-time the data retrieved by the read operation.

Additionally, many web modelling languages allow defining navigation models with content-modification operations (as inserts, updates and deletes) that are executed as a result of browsing a link. As an example, Fig. 2.1 shows that when the user navigates from *NewSale* to *NewSaleLine*, the operation *InsertSale* is executed (using the parameters provided by the user in *NewSale*). The set of modification operations we consider are: *InsertET*(*x*, *v*₁...*v*_n) (resp. *DeleteET*(*x*)) to perform the addition (removal) of an instance *x* into (from) the entity type *ET* (optionally, attributes of *x* may be initialized with values *v*₁...*v*_n), *UpdateA_iET*(*x*, *v*) to set *v* as the new value of the attribute *A_i* in *x* and *InsertRT*(*x*₁, *x*₂) (resp. *DeleteRT*(*x*₁, *x*₂)) to perform the addition (removal) of the fact that objects *x*₁, *x*₂ participate in a link of *RT*.

Some languages admit the definition of more complex operations either as part of the definition of the data model (as in OOWS [7]) or in some additional model (as in [8]). To cope with these operations, all references to a complex operation *op* in the model are replaced with the sequence of basic ones that appear in the definition of *op*.

3. A Graph-based Representation for Navigation Models

We propose a graph-based representation to formally represent navigation models. This representation facilitates an unambiguous definition of our refactorings catalogue. Given a navigation model N , the corresponding graph $G_N = (V_N, A_N)$ is obtained by means of the following rules:

- Every page in N is a vertex in V_N .
- Every link in N from a page X to a page Y becomes an arc from X (i.e. from the vertex representing X in G_N) to Y in A_N .
- The label of a vertex v stores the (possibly empty) set of read operations associated to the page X represented by v in G_N .
- The label of an arc a stores the (possibly empty) ordered sequence of modification operations associated to the link l represented by a in G_N .

Note that G_N is a directed graph (digraph), since being able to navigate from a page X to a page Y does not imply that the navigation from Y to X is also possible. Occasionally, G_N turns out to be a multigraph [9] since it may contain multiple arcs with the same orientation between a pair of vertices v_1 and v_2 . This happens when the page corresponding to v_1 contains several links targeting the page represented by v_2 .

When information about which pages act as home page/s for the web application is available in the input navigation model, the corresponding vertices in the graph are drawn using a dashed line. For the sake of simplicity, information on the internal page layout and structure is not represented in the graph.

Fig. 3.1 shows the graph corresponding to the navigation model of Fig. 2.1.

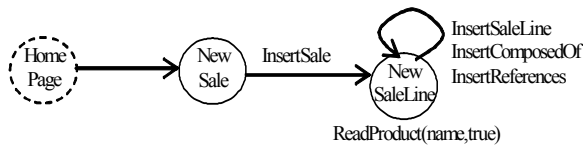


Figure 3.1. Graph definition for the navigation model of Fig. 2.1

We would like to remark that this graph-based representation could be also used to verify basic structural characteristics of the navigation model. For instance, we could check if all pages are reachable from the home page.

Alternative (more complex) representations of navigation models express them by means of statecharts [10] or using Petri-Nets (see [11] as a relevant example).

4. Behaviour Preserving Navigation Models

By definition, a refactoring should never alter the external software behaviour. Unfortunately, a precise definition of this behaviour preserving condition does not exist [2].

The original definition [12] states that, for the same set of input values, the same set of result values must be returned before and after the refactoring. However, this condition may be insufficient depending on the application domain. On the other hand, it may also be too strict since designers may prefer a more pragmatic (though weaker) definition. In this sense, [13] proposes a behaviour preserving condition that does not require to ensure that all combinations of input values generate the same output results after the refactoring but just to check that all method calls are preserved by the refactoring, that is, [13] defines that the behaviour is preserved if the user can execute the same set of methods before and after the refactoring.

According to this more pragmatic view, we define that a refactoring for navigation models is behaviour preserving when the kind of read and modification operations the user may execute is maintained by the refactoring. That is, it may not happen that after the refactoring a user is able to query (or modify) the data of an entity or relationship type that was not previously available (and the other way around, if before the refactoring a user could access/modify a model element, the same kind of access/modification must be allowed in the navigation model after the refactoring). More formally:

Definition 4.1. A refactoring for a navigation model is behaviour preserving when its *read-behaviour* and *update-behaviour* preserving.

Definition 4.2. A refactoring $r(N) \rightarrow N'$ (where N is the initial navigation model and N' the navigation model obtained once the refactoring r is applied over N) is read-behaviour preserving when:

1. For each read operation r appearing in a vertex v , $v \in G_N$, there is a vertex v' , $v' \in G_{N'}$, that includes a read operation r' equal to r .
2. For each read operation r' appearing in a vertex v' , $v' \in G_{N'}$, there is a vertex v , $v \in G_N$, that includes a read operation r equal to r' .

We consider that two read operations r_1 and r_2 are the same operation iff r_1 and r_2 refer to the same entity type and the set of attributes retrieved in r_1 and r_2 coincide. We do not also require that both *selector* conditions are equivalent since when selectors are arbitrary first-order logic predicates, this problem is undecidable.

As an example, a refactoring over our running example (Fig. 3.1) cannot add a read operations over

the customer class. It could neither remove the existing *ReadProduct* operation (but for instance, it could move this operation to a different page).

Definition 4.3. A refactoring $r(N) \rightarrow N'$ is update-behaviour preserving when:

1. For each modification operation op appearing in an arc a , $a \in G_N$, there is an arc a' , $a' \in G_{N'}$, that includes an operation op' equal to op .
2. For each modification operation op' appearing in an arc a' , $a' \in G_{N'}$, there exists an arc a , $a \in G_N$, that includes an operation op equal to op' .

We say that two modification operations are the same operation if they represent the same kind of modification (insert/update/delete) over the same model element. We do not require that the possible set of argument values they may receive as input values coincide. This cannot be determined at design-time since parameter arguments may come from values provided by the user at run-time.

When evaluating both definitions we should only consider the set of vertices (and their arcs) reachable from a home page.

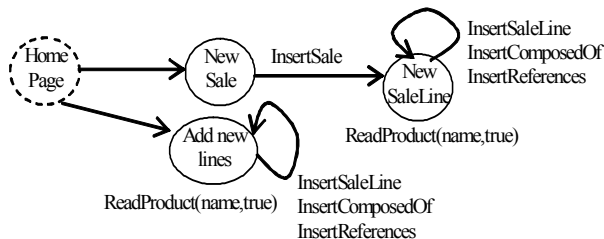


Figure. 4.1. A behaviour preserving refactoring of the navigation model in Fig. 3.1

For instance, a refactoring over our running example could generate the new navigation model shown in Fig. 4.1. This new model contains a new page to add new lines to an existing sale (for the sake of simplicity, we assume that the user directly provides the *id* of the sale, so no read operations on the *Sale* type are needed). The model is update-behaviour preserving (and read-behaviour preserving as well) since all operations in the new links were already included in the original model; the user cannot apply after the refactoring changes on the data not permitted before.

Additionally, in the previous definitions, the refactored navigation model N' must be correct [14]. That is, all possible navigation paths in N' must be capable of leaving the underlying application data in a consistent state (i.e. a state that satisfies all integrity constraints defined in the data model). This can be determined by means of computing all possible navigation paths in N' and checking that the sequence of operations appearing in the arcs of each path can possibly evolve the data to a consistent state. The algorithm to check the correctness of a navigation

model was already presented in [14]. To simplify the presentation of our refactorings, we will omit expliciting this condition in their definition.

We are aware of the trade-offs implicit in our definition of behaviour preserving for navigation models. Stronger conditions to assess behaviour preservation for navigation models (as, for instance, requesting that not only the type of the modification operations but their ordering is preserved after the refactoring as well) could have been stated. However, we prefer to favour flexibility instead of a more strict application of the refactoring process. The study of a wider range of behaviour preserving definitions and their effect on the results of the refactoring process is left as further work.

5. Catalogue of Refactorings

In this section we present a catalogue of refactorings for navigation models. The refactorings can be combined to generate more complex transformation sequences. These refactorings are behaviour-preserving, according to our definition of behaviour preservation given in the previous section. Alternative definitions could result in a different refactorings list.

All refactorings are expressed as graph transformation rules over the graph representing the navigation model. At the end of the refactoring process the resulting graph can be translated back into the actual navigation model by means of reversing the rules introduced in Section 3.

Designers may use these refactorings to improve the navigation model. The exact set of refactorings to apply will depend on the designer's goals. For instance, some refactorings reduce the size of the navigation model (useful to reduce the complexity of the model) while others introduce new model elements, even some redundant ones (which may favour its usability).

Due to lack of space we only provide a partial list of refactorings that covers the basic modifications on pages, links and navigation paths. Thus, for instance, refactorings over operations (to move operations to a different link or to simplify operation sequences taken into account the semantics of each individual operation) are not described.

5.1. Graph transformations

Graph transformation [15] is a popular rule-based technique for expressing model transformations [16] when models are expressed as graphs. In what follows we summarize the main elements of graph transformations in order to facilitate the interpretation of our refactorings. We do not stick to a particular

graph transformation language (see [16] for examples) but use common characteristics of all of them.

Graph transformation rules consist of a LHS (left-hand side) and a RHS (right-hand side) graph patterns. The LHS matches a subset of the source graph. This subset is then modified according to the RHS. Roughly, elements in LHS not included in RHS are removed from the graph while elements in RHS but not in LHS are created. The LHS may contain additional conditions, as negative conditions or textual conditions that restrict the rule applicability. Each rule can be iteratively applied as long as the model still contains a match for the rule.

As an example, Fig. 5.1 shows a graph transformation rule that creates a link between a home page and a page including (at least) one read operation. H , P and $ReadET_i$ are variables of the transformation rule so any subgraph of the model including a home page and a page with a read operation (no matter the entity type queried by the read operation) can be mapped to these variables and be a match for the LHS. Since the only difference between the RHS and the LHS is the link between H and P , the creation of this link is the only change performed by the rule.

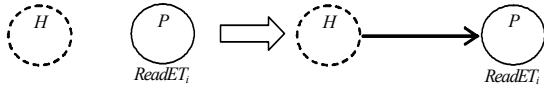


Figure 5.1. A simple graph transformation rule

5.2. Refactorings over links

Add link. This refactoring creates a new link between a pair of pages¹ A and B to provide a direct access to B from A . To ensure the behaviour preservation condition, a new link may be created between two pages either when the link has no attached modification operations (Fig. 5.2. a) or when all operations already appear in some other existing link in the graph (Fig. 5.2. b shows the patterns when the new link has a single attached operation op).

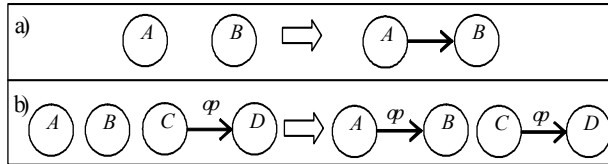


Figure 5.2. Add link refactoring rules

¹ Unless explicitly stated, the proposed refactorings can be also applied over home pages (and their incoming and outgoing links) without variations. Due to lack of space, this is not explicitly shown in the figures that graphically describe the transformation rules.

We could apply this refactoring over our running example (Fig. 3.1) to create a new link between the *NewSale* page and the *HomePage* (Fig. 5.3).

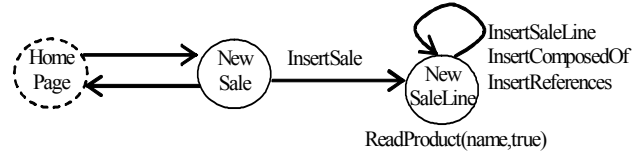


Figure 5.3. Adding a link from *NewSale* page to *HomePage*

Change source page. A link to a page C is moved from a page A to a page B (Fig. 5.4). The link may be labelled. In this and the following patterns, we denote as $sOpXY$ the (possibly empty) sequence of operations attached to a link going from page X to page Y in the LHS. Note that, to preserve the model behaviour, the refactoring does not change the sequence of operations $sOpAC$ attached to the link; the same sequence appears associated to the outgoing link from the new source page B in the RHS. Note that this refactoring preserves the reachability of C .

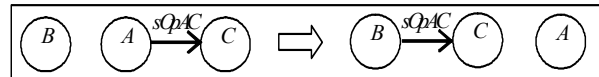


Figure 5.4. Change source page refactoring rule

Change destination page. The target page of a link is changed from a page A to a page B (Fig. 5.5).

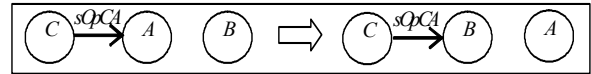


Figure 5.5. Change destination page refactoring rule

Clone link. A link is duplicated. The source and destination pages are not changed (they can be changed afterwards using the above refactorings). All link operations (if any) are also cloned (Fig. 5.6).

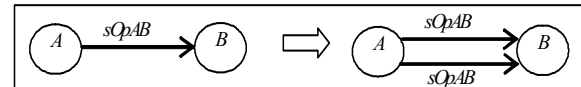


Figure 5.6. Clone link refactoring rule

Remove link. A link is removed. To preserve the reachability of the destination page, this page must have at least another incoming link. To ensure the behaviour preserving condition, the removed link either does not present attached operations (Fig. 5.7 a) or all of them also appear in other links (Fig. 5.7. b shows the LHS and RHS for links with a single attached operation).

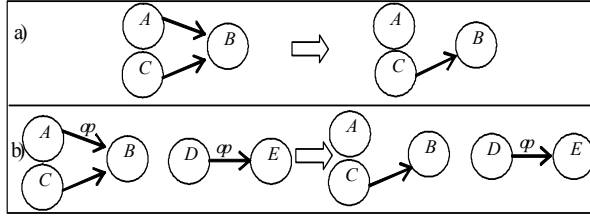


Figure 5.7. Remove link refactoring rules

5.3. Refactorings over pages

Mark as home page. A page A is marked as a new home page (Fig. 5.8 a).

Unmark home page. A home page A is transformed into a “normal” page. This is only possible if the navigation model contains at least another home page (Fig. 5.8 b)

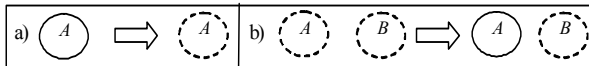


Figure 5.8. Mark as home page and Unmark home page refactoring rules

Add page. A new page B is created in the navigation model. To guarantee its reachability, this new page is linked to an existing page A. To preserve the external model behaviour, the new page must not include read operations (Fig. 5.9 a) or all its read operations must already appear in other existing pages (Fig. 5.9 b shows the rule for new pages with a single read operation over an entity type ET_i).

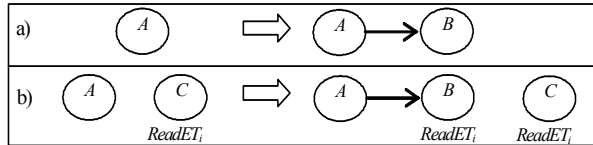


Figure 5.9. Add page refactoring rules

Fig. 5.10 shows the application of this refactoring over our running example. A new *Contact* page is created and linked both ways (the second link is created with the help of the *add link* refactoring) to the *Home Page*.

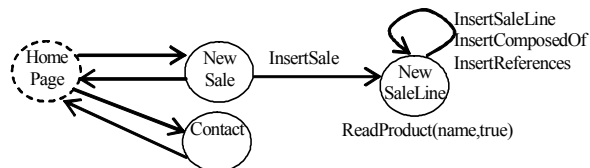


Figure 5.10. Adding a new *Contact* page and a link from this new page to the home page

Clone page. A page is duplicated (including all its read operations). All incoming and outgoing links along

with their modification operations are cloned as well (Fig. 5.11).

Parallel Merge. Two non-consecutive pages P_1 and P_2 are merged into a single one with $sReadP_1 \cup sReadP_2$ read operations and with all P_1 and P_2 incoming and outgoing links (and their corresponding operations). In this and the following refactorings, we denote as $sReadX$ the (possibly empty) set of read operations of a page X (Fig. 5.12).

Sequence Merge. Two consecutive pages P_1 and P_2 are merged into a single one (Fig. 5.13). The resulting page has the P_1 incoming links and the P_2 outgoing links. When the P_1 -to- P_2 link has operations, they are added into all outgoing links of the merged P_1P_2 page. An expression like ‘ $sOp_1 \parallel sOp_2$ ’ in the RHS denotes that the link contains the concatenation of the sequence of operations in sOp_1 plus the sequence of operations in sOp_2 .

Split. A page P is split up into two consecutive pages P_1 and P_2 connected by a simple link. Incoming P links are redirected to P_1 . Outgoing P links become anchored in P_2 . Read operations in P can be moved to P_1 , P_2 or both (Fig. 5.14).

As an example, an application of the split refactoring over the *NewSaleLine* page may generate the *SaleLineData* and *SelectProduct* pages (Fig. 5.15). In the first one, the user enters the values for the attributes of *SaleLine* type (as the *quantity* attribute in our example) while in the second one he/she selects the purchased product (in this case, *ReadProduct* is moved to this second page). After that, the new sale line is added to the database and the process can repeat again. We could also clone the link from *SelectProduct* to *SaleLineData* (*clone link* refactoring) and make it point to *NewSale* (*change destination page* refactoring) so that, after creating all sale lines, we can directly start inserting a new sale.

Remove Page. A page A is removed from the model. All incoming and outgoing links are removed as well. This refactoring can only be applied if all links can be removed according to the conditions stated in the *remove link* refactoring. Besides, if the page has read operations, all read operations must appear in other existing pages in the model. Fig. 5.16 shows the refactoring rules for pages without (a) and with a single read operation (b).

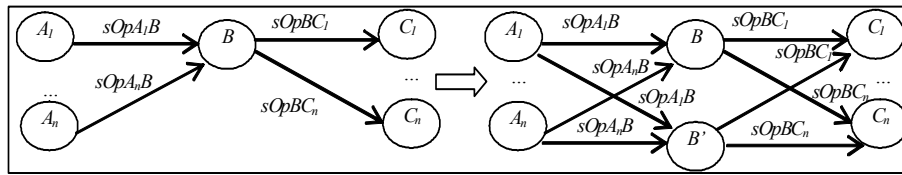


Figure 5.11. Clone page refactoring rule

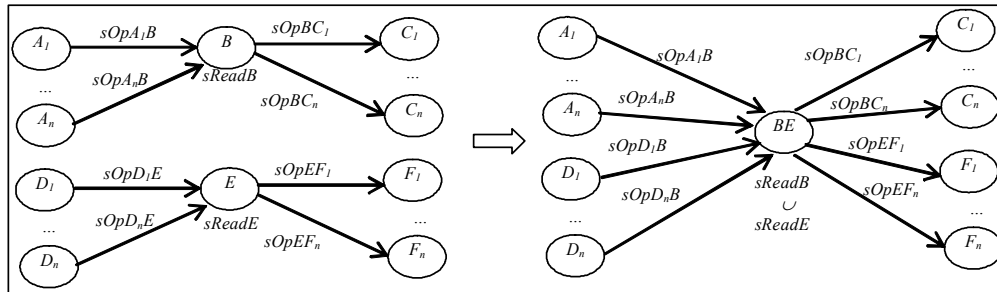


Figure 5.12. Parallel Merge refactoring rule

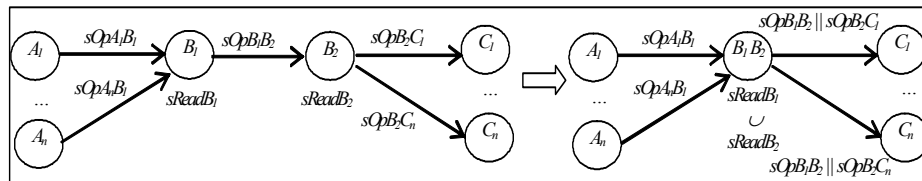


Figure 5.13. Sequence Merge refactoring rule

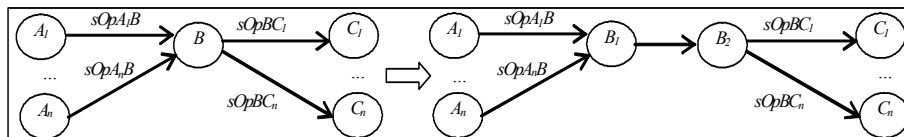


Figure 5.14. Split refactoring rule

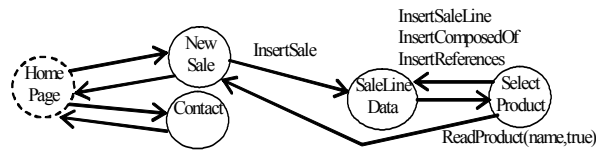


Figure 5.15. NewSaleLine is split into two different pages.

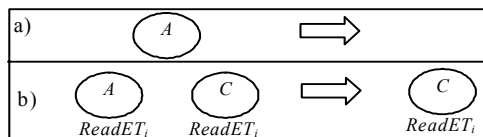


Figure 5.16 Remove page refactoring rules

5.4. Refactorings over Navigation Paths

Remove redundant navigation paths. Redundant paths in the model are usually useless and could be removed to improve the structure of the model. We say that two paths starting in the same home page are redundant if the ordered sequence of modification operations associated to the arcs coincide and the set of read operations attached to the pages of both paths is equivalent. Note that we do not require that both paths have the same number of pages nor that operations appear exactly in the same position, as long as the operation sequences satisfy the previous conditions (Fig. 5.17).

Head-Merge of navigation paths. Two navigation paths with an equivalent beginning part can be merged into a single sequence that divides after the common part ends. We determine that two paths share a common beginning with the same procedure stated in the previous pattern to detect redundant paths. The common part is the one sharing the same sequence of operations and read operations (Fig. 5.18).

Tail-Merge of navigation paths. The common part of two navigation paths presenting an equivalent ending part can be merged (Fig. 5.19).

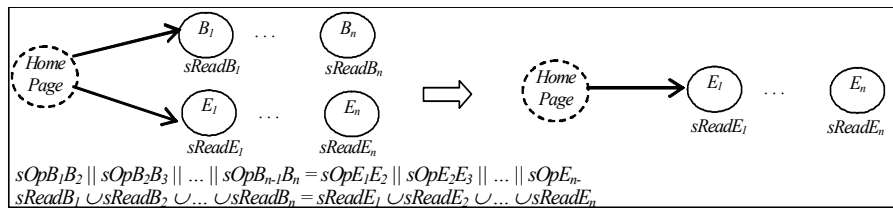


Figure 5.17. Remove redundant path refactoring rule

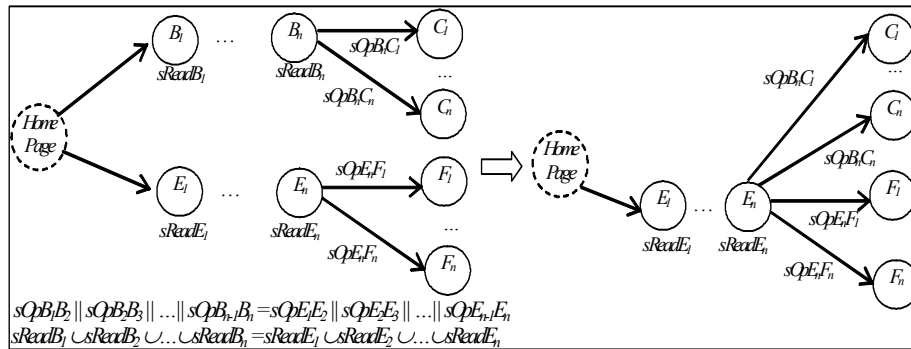


Figure 5.18. Head-Merge refactoring rule

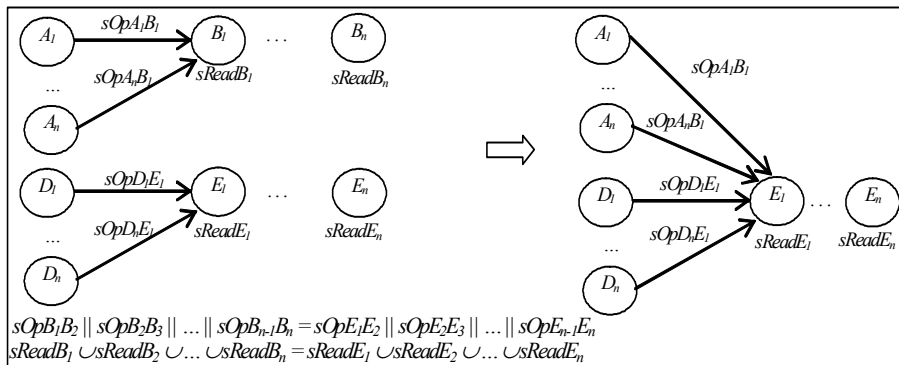


Figure 5.19. Tail-merge refactoring rule

6. Tool Support

We have developed a prototype tool to assist the designer during the application of our refactorings.

In particular, the input of the tool is a WebML model (saved as an XML file) specified with WebRatio [17]. This initial model is processed by the tool and transformed into our graph-based representation. Then, the designer may evolve the

model by selecting one of our refactoring operations. If the refactoring can be applied (i.e. the left-hand side pattern of the refactoring rule has a match on the graph), the tool updates the graph (according to the right-hand side pattern)

Once the designer feels that all necessary refactorings have been performed, the modified model can be exported as an XML file and imported back again into the WebRatio tool to proceed with the development process.

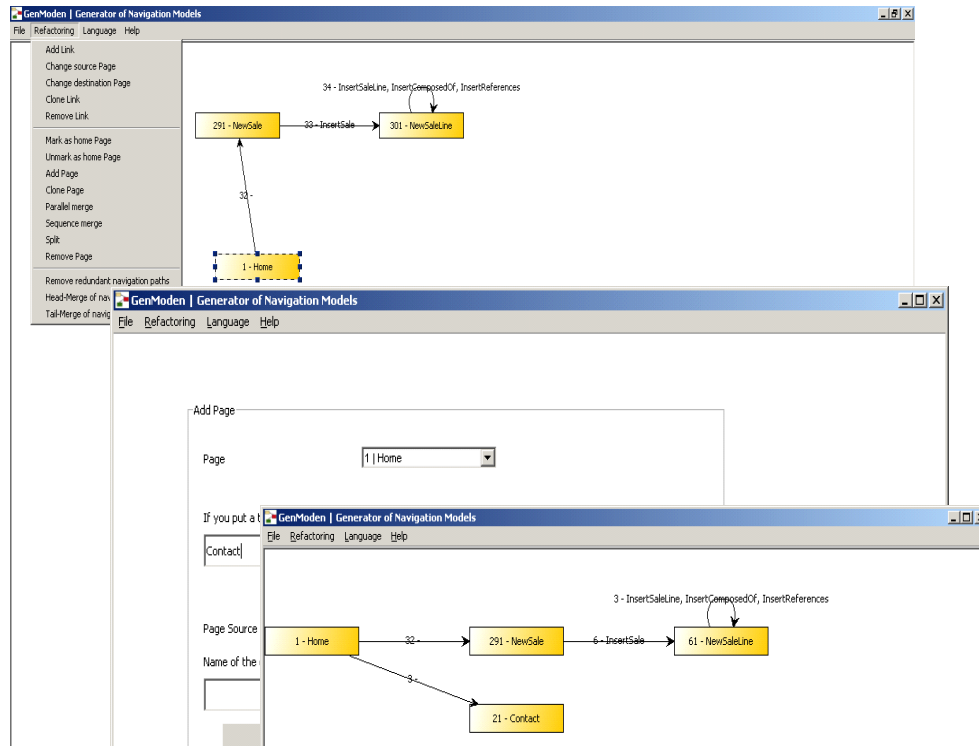


Figure 6.1. Application of the *AddPage* refactoring over an example navigation model

7. Automatic Refactorings Selection

So far we have assumed that the refactorings are manually selected by the designer. However, we envision a more ambitious approach, where refactorings are (partially) selected and suggested to the designer depending on the current structure of the navigation model and the quality goals pursued by the designer (usability, simplicity, minimization of the length of navigation paths and so forth).

A complete description of how such an approach could be realized is out of the scope of this paper and left as further work.

Nevertheless, we would like to, at least, state its main elements:

- A set of metrics to collect representative information of the current quality state of the navigation model. These metrics could be based on the syntactic definition of the model [18] (number of pages, number of links, average number of links per page...) or on run-time information provided by web-mining techniques [19] (most visited pages, most popular navigation paths,...) for those navigation models corresponding to already running applications.
- A list of relevant *<metric,goal>* combinations. That is, for each possible quality goal the designer may choose, we should determine the list of metrics that may help to evaluate the fulfilment of that goal.

- A list of thresholds for each $\langle \text{metric}, \text{goal} \rangle$ combination. These values would serve as an alert to detect and highlight those aspects of the navigation model that do not meet the quality requirements for the goal.
- A list of $\langle \text{refactoring}, \text{metric}, \text{effect} \rangle$ tuples stating the (positive or negative) effect of a refactoring over each metric. The effect may be quantitative or qualitative.

Given these set of elements, an algorithm for selecting the right refactoring combinations to improve a navigation model N according to a goal G could proceed as follows:

1. Evaluate the initial values of N for the relevant metrics of G .
2. Select a metric M from those presenting a value below the defined threshold. The metric could be manually chosen by the designer or randomly selected (and likewise for the following steps).
3. Select a refactoring R among those with a positive effect on M .
4. Use “bad smells” to detect those parts of the navigation model where R could be more effectively applied. A bad smell is a structure in the code (the navigation model in our case) that “suggest (and sometimes scream for) the possibility of refactoring” [1].
5. Apply R .
6. Repeat until all metric values satisfy the threshold.

8. Conclusions and Further Work

We have presented a catalogue of refactorings to improve the quality of existing navigation models. Quality of navigation models is one of the main problems of current web development methods due to their increasing complexity and expressivity.

Our refactorings are formalized as graph transformation rules over a graph-based representation of the given navigation model including read and content-modification operations. Each rule includes the necessary conditions to ensure that the behaviour of the navigation model is preserved by the refactoring.

We plan to continue our work in several directions. First, we would like to expand our list of refactorings by considering also those refactorings on the data model that may affect the related navigation model elements (for instance, the removal of an attribute from an entity type affects all pages reading that attribute) and by admitting more complex navigation models, as the enriched navigation models required to cope with rich internet applications. Secondly, we plan to advance in the automation of the refactoring process,

as sketched in section 7. Finally, we plan to validate our refactorings with an industrial case study.

Acknowledgements

This work was partially supported by the Ministerio de Ciencia y Tecnología and FEDER under project TIN2005-06053.

References

- [1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley Pub Co, 2000.
- [2] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, pp. 126-239, 2004.
- [3] A. Garrido, G. Rossi, and D. Distanto, "Model Refactoring in Web Applications," presented at 9th Int. Symp. on Web Site Evolution (WSE'07), 2007.
- [4] G. Rossi, D. Schwabe, and F. Lyardet, "Improving Web information systems with navigational patterns," *Computer Networks*, vol. 31, pp. 1667-1678, 1999.
- [5] M. v. Welie, "Web Design patterns", <http://www.welie.com/patterns/index.html>, visited May, 2007.
- [6] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera, *Designing Data-Intensive Web Applications*: Morgan Kaufmann, 2002.
- [7] O. Pastor, J. Fons, V. Pelechano, and S. Abrahão, "Conceptual Modelling of Web Applications: The OOWS approach," in *Web Engineering*: Springer-Verlag, 2006, pp. 277-302.
- [8] M. Jakob, H. Schwarz, F. Kaiser, and B. Mitschang, "Modeling and Generating Application Logic for Data-Intensive Web Applications," presented at 6th Int. Conf. on Web Engineering (ICWE'06), 2006.
- [9] B. Bollobás, *Modern Graph Theory*: Springer-Verlag, 1998.
- [10] Karl R. P. H. Leung, Lucas Chi Kwong Hui, S.-M. Yiu, and R. W. M. Tang, "Modeling Web Navigation by Statechart," presented at 24th Int. Conf. on Computer Software and Applications (COMPSAC'00), 2000.
- [11] P. D. Stotts and R. Furuta, "Petri-net-based hypertext: document structure with browsing semantics " *ACM Transactions on Information Systems*, vol. 7, pp. 3-29, 1989.
- [12] W. F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Applicatin Frameworks," University of Illinois, 1992.
- [13] T. Mens, S. Demeyer, and D. Janssens, "Formalizing behaviour preserving program transformation," presented at 1st Int. Conf. on Graph Transformation, 2002.
- [14] J. Cabot and C. Gómez, "On the Quality of Navigation Models with Content-Modification Operations," presented at 7th Int. Conf. on Web Engineering (ICWE'07), 2007.

- [15] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer, "Graph Transformation for Specification and Programming," *Sci. Comput. Program*, vol. 34, pp. 1-54, 1999.
- [16] K. Czarnecki and S. Helsen, "Feature-model-based characterization and survey of model transformation approaches," *IBM Systems Journal*, vol. 45, pp. 621-646, 2006.
- [17] WebModels, "WebRatio." www.webratio.com
- [18] C. Calero, J. Ruiz, and M. Piattini, "A Web Metrics Survey Using WQM," presented at 4th Int. Conf. on Web Engineering (ICWE'04), 2004.
- [19] Q. Zhao, S. S. Bhowmick, and L. Gruenwald, "WAM-Miner: in the search of web access motifs from historical web log data," presented at 14th ACM Int. Conf. on Information and Knowledge Management (CIKM'05), 2005.