Active Extensions in a Visual Interface to Databases

Mariusz Trzaska and Kazimierz Subieta

Polish-Japanese Institute of IT and Institute of Computer Science, Poland.
(mtrzaska, subieta)@pjwstk.edu.pl

Introduction

The retrieval capabilities of the visual information retrieval system Mavigator have been designed for a naive user, typically a computer non-professional. In contrast to retrieval engines on raw text (such as Google), Mavigator addresses structured data (e.g. XML repositories). For such data a query language is proper, however naive users cannot deal with sophisticated retrieval methods and metaphors, especially using keyboard-oriented languages a la SQL and script languages for formatting retrieval output. There are two options: some generic output format (e.g. a table), which is usually too restrictive for the users, or some attractive visual form (e.g. a function chart), which in turn must be specialized to a particular application and retrieval kind. Some tradeoff between these extremes is necessary.

The Active Extensions module, which is a part of the Mavigator prototype [1], allows extending its existing functionalities by professional programmers. In contrast to Visage [2], which uses a dedicated script language, Active Extensions are based on a fully-fledged programming language (C#). Such a solution does not restrict the form of output, execution speed or algorithmic complexity of output formatting functions.

A disadvantage of our solution is that end users asking for a new output format need cooperation with a professional programmer. We believe that this solution is inevitable if we do not want to sacrifice the expressive power of the visual interface. In majority of visual retrieval tasks such a mode of making changes to end user interfaces is acceptable regarding the time, cost and convenience.

The paper is organized as follows. In next section we discuss related work. Then we give detailed description of our proposal concerning new functionalities. After this, Mavigator's information retrieval capabilities

Advances in Information Systems Development: Bridging the Gap between Academia and Industry. Edited by A.G. Nilsson et al., Springer, 2005

are presented. Next section concerns implementation and architecture of a working prototype and the last one concludes.

Related Work

The related solutions could be analysed from two points of views: methods of modifying application's functionalities and the way of information retrieval. There are not so many applications, which could be assigned to the both mentioned groups. Hence we present them in two separate subsections.

Methods of Modifying Application's Functionalities

DRIVE [3] is an example of a user interface to a database development environment. The system dynamically interprets a conceptual object-oriented data language with active constructs. Specification of the interface is made in a textual language called NOODL. The framework includes the following main classes: user, data, interface, and visualisation. Due to separation of data and interfaces, each data item could have associated multiple interface components. Each user could have own set of user-specific views and access privileges. Visual programming facilities help in creating queries, constraints, and other retrieval options. Although DRIVE has been designed as an easy-to-use graphical development system, it is disputable if every user kind (especially a naive one) will be ready to accept it.

Teallach [4] employs the idea of a Model-Based User Interface Development Environment (MB-UIDE). It particularly supports specification of Domain, Task and Presentation Models. A domain model is extracted from a database scheme. Then, using a graphical editor, the user builds an interface by linking together appropriate items from presentation and task models. Teallach does not introduce built-in information retrieval capabilities, thus all retrieval methods should be designed by the user. From one point of view it is an advantage because the user has full freedom in employing various retrieval metaphors. On the other hand, it could be a disadvantage, because there is no common and coherent basis of information retrieval methods.

Visage [2] is an example of another approach. A user interface itself contains some navigational methods for retrieval. Moreover, each data visualization component, called *frame*, could be modified by attaching a special script. Similarly to Mavigator, scripts are written by programmers.

However, in contrast to our approach, Visage utilizes a scripting language similar to Basic. Unfortunately the interpretation overhead limits the data-set size that can be manipulated with no speed constraint. That is one of the reasons for using in Mavigator a fully-fledged programming language.

Visual Tools for Information Retrieval

Roughly speaking, visual metaphors for information retrieval can be subdivided into two groups: based on a graphical query language and based on browsing. Some systems combine features from both groups. An example is Pesto [5] having possibilities to browse through objects from a database. Unlike Mavigator, browsing is performed from one object to a next one. Besides browsing, Pesto supports quite powerful query capabilities. It utilizes a query-in-place feature, which enables the user to access nested objects, e.g. courses of particular students, but still in the one-by-one mode. Another advantage concerns complex queries with the use of existential and universal quantification, however, not very usable for less professional users.

An essential issue behind such interfaces is how the user uses and accumulates information during querying. In particular, the user may see all the attributes even those, which are not required for the current task. Otherwise, the user can hide non-interesting attributes, but this requires from him/her some extra action. Therefore, from the user point of view, there is some tradeoff between actions preparing the information for querying and actions of further querying. To accomplish complex queries the system should support combinations of both types of actions.

Typical visual querying systems are Kaleidoscape [6], based on its language Kaleidoquery, and VOODOO [7]. Both are declared to be visual counterparts of ODMG OQL thus graphical queries are translated to their textual counterparts and then processed by an already implemented query engine.

An example of a browsing system is GOOVI [8]. A strong point of the system is the ability to work with heterogeneous data sources. Another interesting browser is presented in [9], which is dedicated to Criminal Intelligence Analysis. It is based on an object graph and provides facilities to make various analyses. Some of them are: retrieving all objects connected directly/indirectly to specified objects (i.e. e. all people, who are connected to a suspected man), finding similar objects, etc. Querying capabilities include filtering based on attributes and filter patterns. The latter allow filtering links in a valid path by their name, associated type, direction or combi-

nation of these methods. The browsing style is similar to our *extensional navigation*.

Active Extensions

When we started to develop methods of extending existing functionalities two different approaches come to our minds:

- Utilizing some kind of a graphical metaphor like in [3] or [4]. Both of them are tradeoffs between the power and easy-to-use. They are claimed to be easy enough for the target user. In our opinion, however, for our target user they are still too complex. Moreover, the metaphors seriously restrict the field of user retrieval activities.
- Using a programming language. Depending on a language kind, limitations can be reduced partly or at all. We have assumed that a Mavigator's user is not a programmer and will not be able to use such extensions. Hence the support from a professional programmer is required.

Mavigator already employs some information retrieval metaphors (see next section), which are powerful and yet easy-to-use, so we have decided to provide a way to add new functionalities operating only on a query result. The approach does not complicate the entire application's architecture, but guarantees sufficient flexibility.

Mavigator is our second prototype. The first one, called Structural Knowledge Graph Navigator (SKGN), has been designed and implemented (in Java) for the European project ICONS, thus we have gathered some practical experience of its use by computer non-professionals [10], [11], [1]. The current prototype uses Microsoft C# as a language for active extensions. A programmer is aware of the Mavigator meta-data environment, which allows him/her to write a source code of the required functionality in C#. Writing an Active Extension source code is possible through a Mavigator's special editor. Once programmer compiles the code, a particular Active Extension is ready to use (without stopping Mavigator). Then the end user is supported with one click button causing execution of the written code. The code processes a query result or objects recorded in a user basket. The functionality of such programs is unlimited. Next three sub-sections present its particular applications.

Simple Active Extensions

A simplest type of Active Extensions may perform some calculations. In Mavigator we have implemented popular aggregate functions, such as the minimal attribute's value, maximum attribute's value and average attribute's value. All are very easy for use. The user has to select a particular type of calculation and then to select a particular attribute in a query result. Then the result of the calculation is shown to the user.

Active Projections

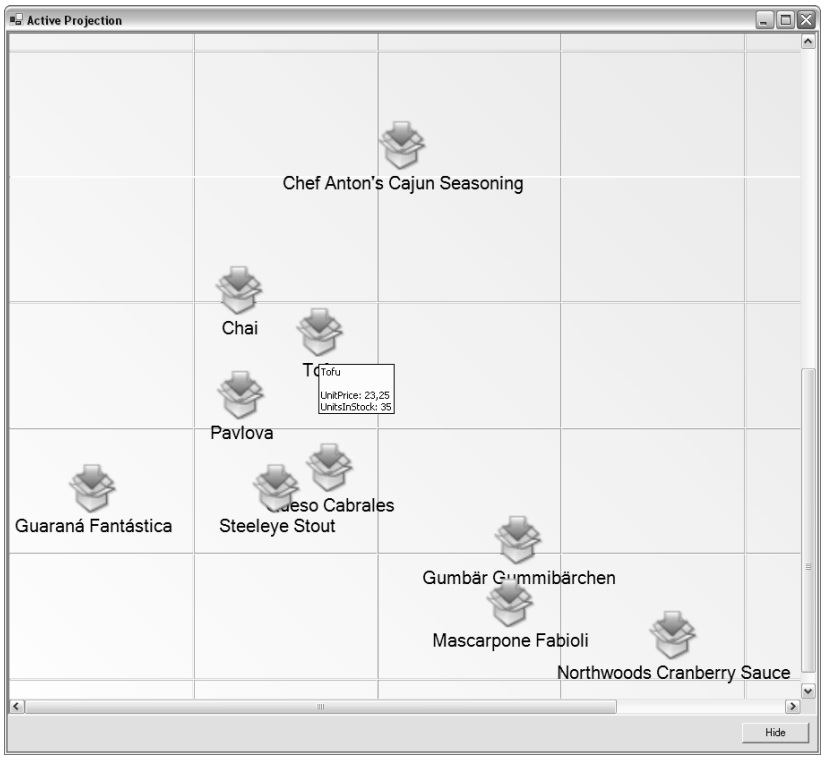

Fig. 1. Active projections

Another application of the Active Extensions module is an active projection (Fig 1) which allows the user to visualize a set of objects. The x, y coordinates of icons representing objects are determined by values of ob-

jects' attributes. The current implementation uses two axes (2D), which allow visualizing dependencies of two attributes. Fig. 1 shows objects of the class Product and theirs dependencies between unit's price and units in stock.

An active projection makes it possible to perform some data mining investigations, in particular, to identify some groups of objects. For instance it is easy to see in Fig. 1 two groups, where one includes cheap products with a higher stock and the second one (right-bottom corner) more expensive products with a smaller stock. One can also observe that there are more cheap products than expensive ones.

Besides the visual analysis of objects dependencies it is also possible to utilize projections in more active fashion. Object taken from a basket can be dropped on projection's surface, which cause right (based on attributes values) placement. It is also possible to perform reverse action: drag an object from the surface onto the basket (which cause recording object in a basket).

Objects Exporters

Objects exporters allow cooperating with other software systems. Having a query result, it is possible to send it to other programs, such as Excel, Crystal Reports, etc. That approach makes it possible a subsequent processing of Mavigator's results of querying/browsing. The current prototype exports to XML files, which could be post-processed by a number of modern software tools.

Information Retrieval Capabilities

Mavigator is made up of three metaphors utilized for information retrieval: intensional navigation, extensional navigation and persistent baskets. The subdivision of graphical querying to "intensional" and "extensional" can be found in [12]. We have adopted these terms for the paradigm based on navigation in a graph. The user can combine these metaphors in an arbitrary way to accomplish a specific task.

Intensional and extensional navigation are based on navigation in a graph according to semantic associations among objects. Because a schema graph (usually dozens of nodes) is much smaller than a corresponding object graph (possibly millions of nodes), we anticipate that intensional navigation will be used as a basic retrieval method, while extensional navigation will be auxiliary and used primarily to refine the results.

Next subsections contain short description of the methods (for detailed one see [1]).

Intensional Navigation

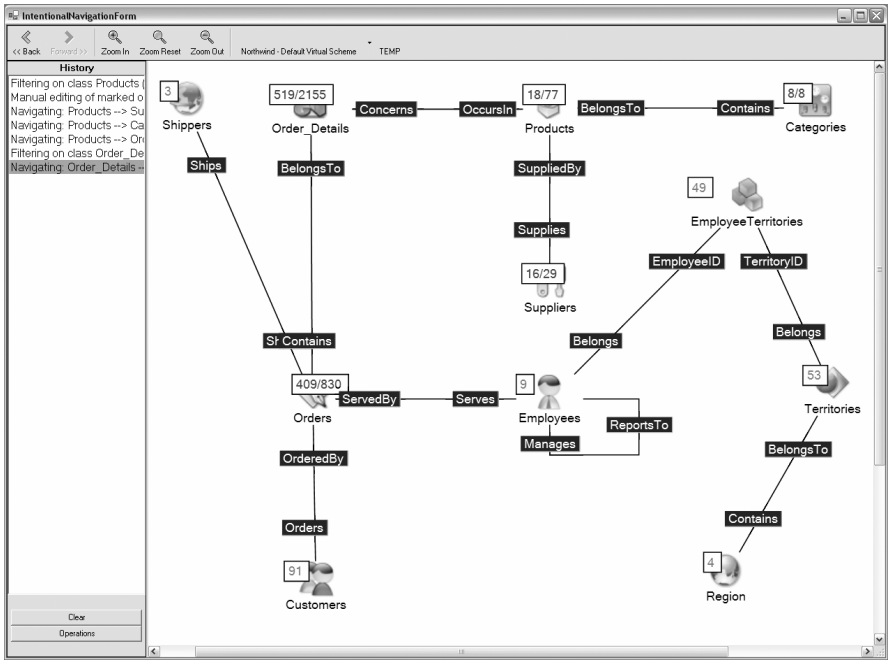

Fig. 2. Intensional navigation window

Intensional navigation utilizes a database schema graph. Fig 2 shows a window containing a database schema graph of the Northwind sample (shipped with the MS SQL Server). A graph consists of the following primitives:

- Vertices, which represent classes or collections of objects. With each of them we associate two numbers: the number of objects that are marked by the user (see further) and the number of all objects in the class,
- Edges, which represent semantic associations among objects (in the UML terms),
- Labels with names of association roles. They are understood as pointers from objects to objects (like in the ODMG standard, C++ binding).

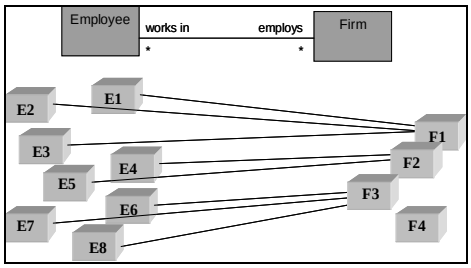

Fig. 3. Explanation of marking objects using intensional navigation

The user can navigate through vertices via edges. Objects that are relevant for the user (candidates to the search result) can be marked, i.e. added to the group of marked objects. There are a number of actions, which cause objects to be marked:

- Filtering through a predicate based on objects' attributes. The action causes marking those objects for which the corresponding predicate is true.
- Manual selection. Using values of special attributes from objects (identifying objects by comprehensive phrases) it is possible to mark particular objects manually. It is especially useful when the number of objects is not too large and there are no common properties among them.
- Navigation (Fig 3) from marked objects of one class, through a selected association role, to objects of another class. An object from a target class is marked if there is an association link to the object from a marked object in the source class. Fig 3 explains the idea. Let's assume that the *Firm* set of marked objects has four marked objects: *F1*,…,*F4*. Than, navigating from *Firm* via *employs* causes marking eight objects: *E1*,…,*E8*. This activity is similar to using path expressions in query languages. A new set of marked objects (the result of navigation) replaces an existing one. It is also possible to perform a union or intersection of new marked objects with the old ones.
- Basket activities – see section about baskets.
- Active extensions. In principle, this capability is introduced to process marked objects rather than to mark objects. However, because all the information on marked objects is accessible from an Active Extension source code, the capability can also be used to mark objects.

Intensional navigation and its features allow the user to receive (in many steps but in a simple way) the same effects as through complex, nested queries. Integrating these methods with extensional navigation, manual se-

lection and other options supports the user even with the power not available in typical query languages.

An open issue concerns functionalities that are available in typical query languages, such as queries involving joins and aggregate functions. There are no technical problems to introduce them to Mavigator (except some extra implementation effort) but we want to avoid situation when excess options will cause our interface to be too complex for the users. We hope that during evaluation we will find answers on such questions.

Extensional Navigation

Extensional navigation takes place inside extensions of classes. Graph's vertices represent objects, and graph's edges represent links. When the user double clicks on a vertex, an appropriate neighborhood (objects and links) is downloaded from the database, which means "growing" of the graph.

Extensional navigation is useful when there are no common rules (or they are hard to define) among required objects. In such a situation the user can start navigation from any related object, and then follow the links. It is possible to use basket for storing temporary objects or to use them as starting points for the navigation.

Baskets

Baskets are persistent storages of search results. They store two kinds of entities: unique object identifiers (OIDs) and sub-baskets. The hierarchy of baskets is especially useful for information categorization and keeping order. Each basket has its name that is typed in by the user during its creation. The user is also not aware of OIDs, because special objects labels are used. During both kinds of navigation, it is possible to drag an object (or a set of marked objects) and to drop them onto a basket. The main basket (holding all the OIDs and sub-baskets) is assigned to a particular user. At the end of a user session, all baskets are stored in the database.

Basket activities include: creating a new basket, removing selected items (sub-baskets or objects), performing operations on two baskets (sum of baskets, intersection of baskets, and set-theoretic difference of baskets; the operation result can be stored in one of the participating baskets or in a new one). There are also two more advanced operations:

- Drag an object and drop it onto an extensional navigation frame. As the result, the neighborhood (other objects and links) of a dropped object will be downloaded from the database.
- Drag a basket and drop it onto class's visualization in the intensional navigation window. As the result, a new set of marked object can be created. Only objects of that class are considered.

Software Architecture and Implementation

The Mavigator prototype is implemented as a Windows Form Application in the C# language. Its architecture (Fig 4) consists of the following elements:

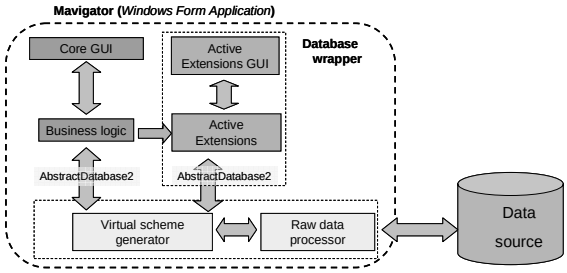

Fig. 4. Architecture of the Mavigator prototype

- Core GUI – contains implementation of the core user interface elements like intensional navigation window, basket window, etc.,
- Business logic – includes implementation of the Mavigator retrieval metaphors and some additional routines,
- Active Extensions GUI – GUI elements being a part of Active Extensions like an Active Projections window,
- Active Extensions – elements compiled from a source code written by an Active Extensions programmer. The arrow, which comes from the business logic block, symbolizes query results processed by AE,
- A database wrapper – ensures communication, via the defined AbstractDatabase2 interface, with any data source. We note that all internal data processing (including Active Extensions) works on an abstract data model (independent of implementation), which ensures that an entire application can work in the same manner aside of the current (possibly,

heterogeneous) data sources. Moreover, an entire application works with virtual schemas. They allow to redefine (using the SBQL query language [[13]]) a physically database scheme. This option can be useful for security, hiding some parts of data, changing data names, and so on.

- Data sources. Currently we are working with an ODRA prototype database, however after implementing a dedicated wrapper it is possible to work with any kind of data source, including object/relational databases, XML/RDF files and repositories, ODBC, JDBC, etc.

The Mavigator prototype utilizes active extensions written in Microsoft C#. The functionality requires compiling and running a source code (which implements a particular extension) during execution (runtime) of the Mavigator. Our first idea was to define some programming interface implemented by a particular C# class created by the programmer. However, finally we have found that such a solution would be too heavy with respect to the goal. The programmer developing a particular Active Extension has to create only one method (in a special class): public, static, with two parameters: an instance of a data wrapper and a collection containing OIDs of the objects being processed. Of course, inside the method could be any valid C# code including calling other modules, creating objects, etc. After successful compilation, the system adds this method to the list of created extensions. When the user wishes to run a particular Active Extension, the system runs an associated method, passing an instance of the data wrapper and a collection of objects' OIDs as parameters.

Conclusions and Future Work

We have presented Mavigator, which offers new qualities in extending existing application's functionalities. The Active Extensions, which use a fully-fledged programming language, make it possible to create any kind of additions to the Mavigator's core functions. The designed architecture is flexible and allows the users to work with any kind of a data source. The utilized data retrieval metaphors are easy to understand for casual (naive) users.

We plan to conduct a formal usability test on a group of users. We have some, generally positive, informal input coming from the users of the ICONS prototype. We also plan investigations concerning new visual functionalities and metaphors, which will make our tool more powerful and easy-to-use.

References

[1] Trzaska M, Subieta K (2004) Usability of Visual Information Retrieval Metaphors for Object-Oriented Databases. Proc of the On The Move Federated Conferences and Workshops (DOA, ODBASE, CoopIS, PhD Symposium) Springer Lecture Notes in Computer Science (LNCS 3292)

[2] Roth F, Chuah M, Kerpedjiev S, Kolojejchick J, Lucas P (1997) Towards an Information Visualization Workspace: Combining Multiple Means of Expression. Human-Computer Interaction Journal, vol 12, no 1 & 2, pp 131-185

[3] Mitchell K, Kennedy J (1996) DRIVE: An Environment for the Organised Construction of User Interfaces to Databases. 3rd International Workshop on Interfaces to Databases. Springer-Verlag Electronic WIC

[4] Barclay PJ, Griffiths T, McKirdy J, Kennedy J, Cooper R, Paton NW, Gray P (2003) Teallach – A Flexible User-Interface Development Environment for Object Database Applications. Journal of Visual Languages and Computing, vol 14, no 1, pp 47-77

[5] Carey MJ, Haas LM, Maganty V, Williams JH (1996) PESTO: An Integrated Query/Browser for Object Databases. Proc of VLDB

[6] Murray N, Goble C, Paton N (1998) Kaleidoscape: A 3D Environment for Querying ODMG Compliant Databases. Proc of Visual Databases 4 L'Aquila Italy 27-29 May

[7] Fegaras L (1999) VOODOO: A Visual Object-Oriented Database Language for ODMG OQL. ECOOP Workshop on Object-Oriented Databases 1999

[8] Cassel K, Risch T (2001) An Object-Oriented Multi-Mediator Browser. 2nd International Workshop on User Interfaces to Data Intensive Systems. Zürich Switzerland

[9] Smith M, King P (2002) The Exploratory Construction Of Database Views. Research Report BBKCS-02-02 School of CS and IS Birkbeck College University of London

[10] Trzaska M, Subieta K (2004) Structural Knowledge Graph Navigator for the Icons Prototype. Proc of the IASTED International Conference on Databases and Applications (DBA 2004)

[11] Trzaska M, Subieta K (2004) The User as Navigator. Proc of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS) September 2004 Budapest Hungary

[12] Batini C, Catarci T, Costabile MF, Levialdi S (1991) Visual Strategies for Querying Databases. Proc of the IEEE International Workshop on Visual Languages. Japan

[13] Subieta K, Beeri C, Matthes F, Schmidt JW (1995) A Stack-Based Approach to Query Languages. Proc of the 2nd East-West Database Workshop, Springer Workshops in Computing