Polish-Japanese Institute of Information Technology

Mariusz Trzaska

Usability of Visual Information Retrieval Metaphors for Object-Oriented Databases

Ph.D. Thesis

Submitted to the Senate of the Polish-Japanese Institute of Information Technology

Advisor:

Prof. Kazimierz Subieta

Warsaw, July 2005

♥ to my Family ♥

Abstract

The methods for information retrieval must be adequate to a kind of data that are to be queried, as well as to a kind of target users. This observation is especially important in case of naive users (computer non-professionals), who are not able (or just do not want) to learn sophisticated methods requiring significant learning effort and high computer education. In contrast to retrieval engines addressing raw text data such as Google (that are indeed very easy for naive users), a lot of novel technologies address structured data (e.g. XML/RDF repositories or object-oriented databases). Thus there is a need for user interfaces allowing querying and browsing such structured data. However naive users cannot deal with sophisticated retrieval methods and metaphors, especially using keyboard-oriented languages such as SQL, OQL or XQuery, and script languages for formatting retrieval output. Usually, such users prefer working with some kind of a user-friendly visual interface. The problem is amplified by the fact that the amount of data to work with and nowadays users' expectations regarding application's functionalities and easiness of use, are much bigger than in the past.

In this dissertation we propose a set of graphical metaphors, which are the result of our investigations into easy in use and yet powerful visual querying and browsing capabilities. The research has been done on the basis of a working prototype called Mavigator. The basic research thesis that we want to promote is that such an interface must solve five basic issues:

- How to present to the user the data that are to be queried or browsed, and how to limit the user view only to those data that are pertinent to his/her current interests?
- Which graphical metaphors are relevant for querying and browsing the data and how the result of the user actions is to be recorded?
- How the result of the querying and browsing is to be presented to the user?
- How the user interface could support user awareness, i.e. how to reduce the danger that the user will be lost after some actions that he/she had performed?
- How to extend existing application's functionality?

All the presented issues are critical in the sense of *usability*, i.e. ability of the interface to be accepted by a wide community of end users. It is likely that neglecting any of the above issues will undermine the sense of building such an interface.

Our implemented solution allows us to make general conclusions concerning this kind of user interfaces with respect of all of the above issues. Because database (XML, RDF) schemas tend to be very complex, the first general thesis is that the user should have the possibility to reduce and customize his/her view on data that are to be queried. We propose the Virtual Schemas module that allows the user to customize database schema, in particular, to change data names, to add new associations, to remove some attributes or classes, etc. The solution is based on views defined in the query language SBQL. As information retrieval and browsing visual metaphors we have assumed intensional navigation (in a schema graph), extensional navigation (in an object graph) and storing intermediate and final querying/browsing results in persistent annotated baskets. The Mavigator's Active Extensions module solves the third of the above issues. It allows the programmer to extend ad hoc existing core functionalities (in particular, to display the retrieval result in any graphical form) through a fully-fledged programming language (in Mavigator - C#). Concerning user awareness we have proposed several methods such as undo functions, recording the history of user work, etc.

The proposed solutions are compared with the state of the art where we have shown that the problem of easy-to-use graphical interfaces is extensively investigated and that we offer new solutions.

CONTENTS

1 Introduction

Databases, among other features, should provide easy access to information. However, the “easiness” is relative to a user kind and his/her experience in computer technologies. Nowadays more and more non-professionals use computers, especially various applications based on Web browser-oriented interfaces. Their requirements with respect to user-friendliness of the entire computer systems are much stronger than requirements of computer professionals. Non-professionals usually do not accept formalized syntax, strict sophisticated rules of user action, long and specialized training or following professional manuals. As a consequence, computer tools, including interfaces for information retrieval, must evolve into non-sophisticated and easy-to-use applications.

On the other hand, the interface should be simple, but not simpler than the inherent complexity of the task that the user has to solve. For some tasks the typical full-text retrieval (as e.g. in Google) is not sufficiently precise because of lack of facilities to specify semantic meaning of data items stored on Web. The XML technologies (including RDF, OWL, Semantic Web and other proposals) aim at putting data into nested labeled structures with well-defined semantic meaning. There are also query languages such as XQuery to express the user needs much more precisely than it is possible in full-text retrieval engines.

In the database domain there are many query languages proposed, in particular, SQL as a major language for accessing relational databases. SQL, like all textual query languages, is very powerful, but its audience, due to its complexity, is rather limited to computer professionals. The same concerns more recent textual languages for object-oriented databases, such as OQL [ODM00] or SBQL [Sub95].

Attempts to define more user-friendly interfaces have been noticed for many years. One of the most successful was QBE [Zlo77] based on a tabular view of relational tables and specific retrieval conditions inserted by the user into the tables. QBE was perhaps the first graphical query interface based on visualization of a database schema and specific visual manipulations of the user on the graphical interface. Due to better hardware and popularity of easy application programming interfaces for graphical manipulations we observe recently extensive development of visual metaphors for

information retrieval. Some of them are counterparts to their textual predecessors. Other, like Value Bars [Chi92] and the interface presented in [Kum97] are based on specific graphical ideas.

As noted in [Cat00], the key issue behind such proposals is *usability* in real applications prepared for users who are not computer professionals. Unfortunately, usability cannot be predicted in advance during design and implementation of an interface, because it depends on many factors such as the readiness of the user to get some training, inherent complexity of the task that he/she has to accomplish and adequacy of the interface to this task, supporting various forms of user awareness during long sessions, and others. Therefore the only way to check the usability is to implement a particular metaphor and then, to measure some user-oriented factors (such as the number of errors, the entire time to reach the goal, etc.) in real applications. The result of our surveys has been described in Chapter 6.

In our opinion, an ideal graphical user interface for naïve users should be characteristic of:

- Graphical (visual) metaphor which will be easy to understand, use and assures the power enough to perform the required work. It could be some kind of a query mechanism, a browsing tool or a solution based on both approaches,
- Possibility to modify some aspect of the tool by adding plug-ins. Depending on assumptions made, those modifications could concern various aspect of the application. However, we do believe that, our target user – computer non-professional, is not able to do it personally, especially if we do not want to sacrifice the power of the solution. Hence, we propose an approach, where a naive user collaborates with a programmer.

As an incarnation of our postulates we introduce the visual querying interface Mavigator, which has been developed to allow computer non-professionals to work with object-oriented database systems. Its key concepts include: intensional navigation, extensional navigation, persistent baskets for recording temporary and final results of querying, virtual schemas and active extensions. The first kind of navigation (intensional) is based on navigation in a database schema graph. The user moves from

vertex to vertex via edges, where vertices denote classes (more precisely, collections of objects) and edges denote associations among classes (in the UML terms). Additional functionalities allow the user to build quite complex queries. The second kind of navigation (extensional) is similar but the user navigates in a graph of objects rather than in a graph of classes. This mode enables the user to move (in mind) from an object to an object (vertices of the graph) via a specific link (an edge of the graph).

Both kinds of navigations heavily deal with database schema graph. Therefore we introduce virtual schemas, which are some kind of database views on the application side rather than database itself. They allow to virtually redefine (in terms of hiding classes, creating attributes, associations and changing schema's names) database schema graph. All users' activities related to the database schema graph proceed transparently on virtual schemas.

After the user has retrieved some result he/she may require to present it visually in some friendly way, e.g. as a tabular report, as a chart, as a distribution map, etc. This is a critical issue for visual querying interfaces, as the trade-off between simplicity of the end user interface and complexity of a possible visualization form perhaps does not exist. The number of options and the general complexity of the interface that may be required to visualize a querying result seem to be unacceptable for naive users. Therefore for achieving the required goal we assume some contribution of computer professionals. The last feature, called active extensions, allows a computer professional to add functionalities required by a particular naive user. We assume that the functionalities will be coded in a programming language (currently C#) thus the range of the functionalities is unlimited.

The objectives of this dissertation are the following:

- Introduce visual metaphors which will be easy to use yet powerful for naive users (computer non-professionals),
- research on the querying and browsing environment which will allow:
    - Adding a new functionality (especially for processing query results),
    - Virtual modifications of an existing database schema graph (without physical modification of the database structure), the user works with, expressed in the database query language SBQL,

- Implementing a prototype that will make it possible to conduct research on the usability of such a solution.

2 Related Research and Solutions

The goal of this chapter is to provide an overview of ideas and prototypes related to the visual information retrieval systems. At the beginning there is a short introduction to the problem, than the detailed description of particular groups of systems is given. However, to our best knowledge, there is no system, which brings together all properties offered by us:

- Easy yet powerful visual metaphors,
- Flexible methods for adding functionalities,
- Working with a virtual schema rather than with a physical database schema itself.

We will describe particular systems by giving their properties and showing similarities and differences to our approach.

2.1 Visual Information Retrieval Capabilities

Roughly speaking, visual metaphors for information retrieval can be subdivided into two groups: based on graphical query languages and based on graphical browsing interfaces. This subdivision is not fully precise because some systems have features from both groups. An example is Pesto [Car96] (see Figure 1) having possibilities to browse through objects from a database. Otherwise to Mavigator, the browsing is performed from one object to a next one object. For instance, the user can display a Student class object, but to see another student, he/she needs to click next (or previous) button and replace the current visualization. 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. Such complex features may however compromise usability for some kinds of users and kinds of retrieval tasks.

In our opinion an essential issue behind such interfaces is how the user uses and accumulates information during querying. For instance, 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 that have to prepare the information necessary for querying and actions of further querying. To accomplish complex queries without putting them explicitly the system should support any sequences of both types of actions.

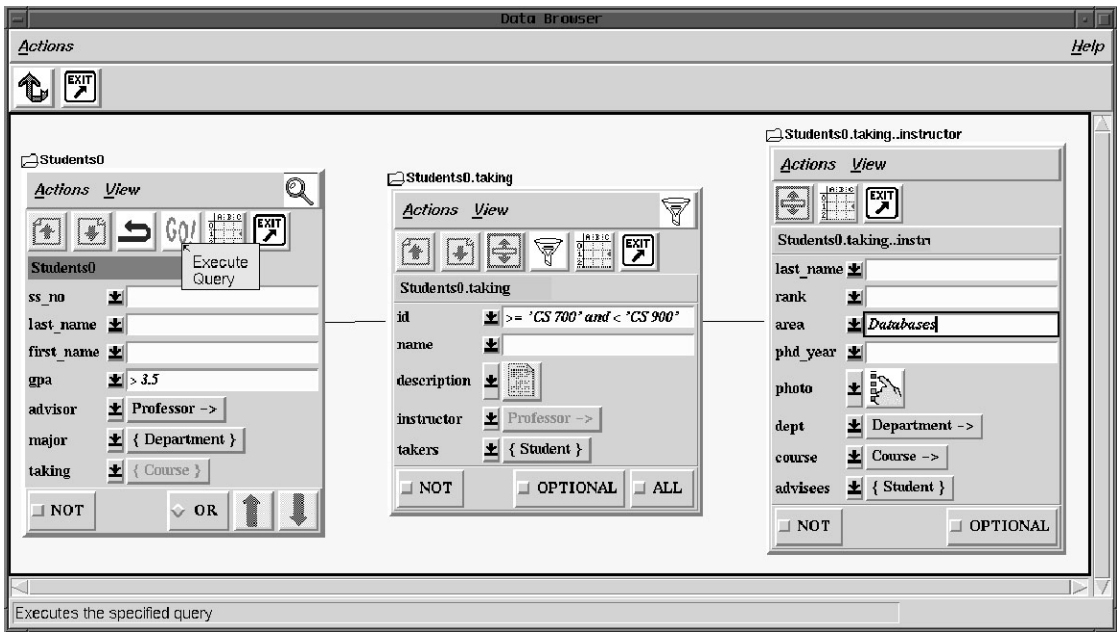

Figure 1. PESTO and its query-in-place

A typical visual querying system is VOODOO [Fer99] (which means Visual Object-Oriented Database language for ODMG OQL [ODM00]). A query is build as a tree containing classes in nodes called templates. Particular nodes are expanded only if some information should be displayed. A query is defined by mapping templates into values using a graphical user interface containing buttons, menus, etc. Than the query is translated to its textual counterpart and then processed by an already implemented query engine. A strong type checking system guarantees that only valid queries are formulated. For instance if we would like to know the name of the department whose head is Smith, we could write OQL query shown in Example 1. An equivalent visual query formulated in VOODOO is shown on Figure 2.

```
select name: d.name
from d in departments
where d.head.name = “Smith”
```

Example 1. OQL query selecting the name of the department whose head is Smith.

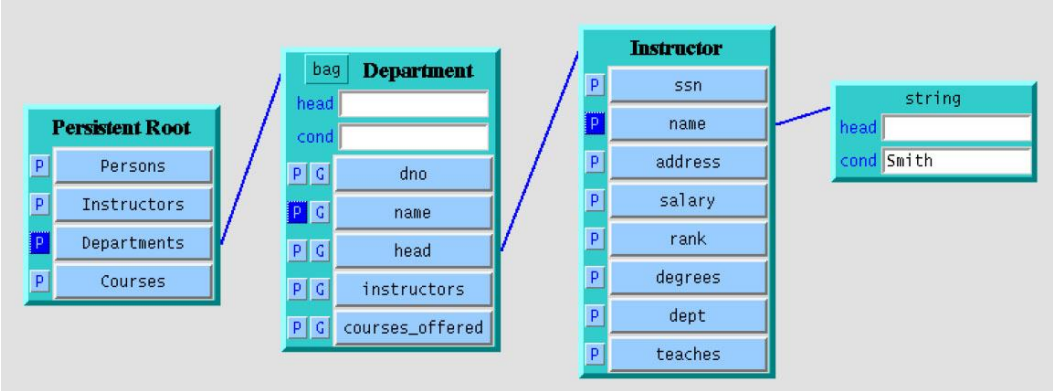

Figure 2. VOODOO query selecting the name of the department whose head is Smith.

VOODOO capabilities are not restricted to such a simple queries. It is also possible to use more complex ones, for instance, find names and addresses of all instructors in the CSE department who earn more than 100 000. An appropriate OQL code is presented in Example 2 and its visual counterpart is shown in Figure 3.

```
select name: e.name, address: e.address
from d in departments,
     e in d.instructors
where d.name = “CSE”
  and e.salary > 100000
```

Example 2. OQL query finding names and addresses of all instructors in CSE department who earn more than 100 000.

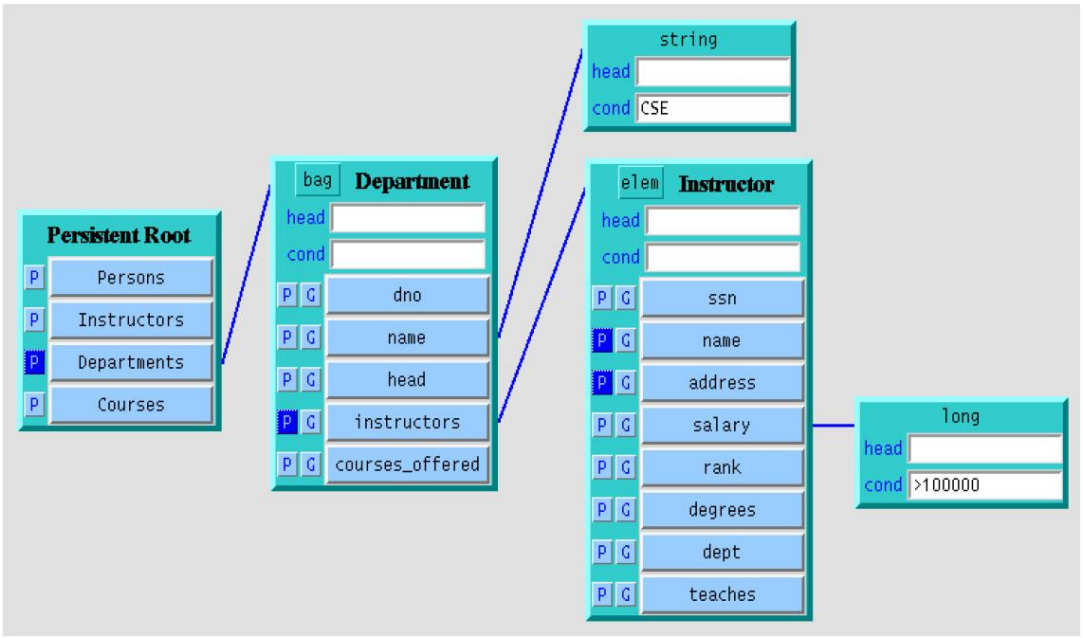

Figure 3. VOODOO query finding names and addresses of all instructors in CSE department who earn more than 100 000.

Another visual query system is Kaleidoscape [Mur98] (Figure 4), based on its visual language Kaleidoquery [Mur00]. Similarly to VOODOO it is declared to be visual counterpart of ODMG OQL.

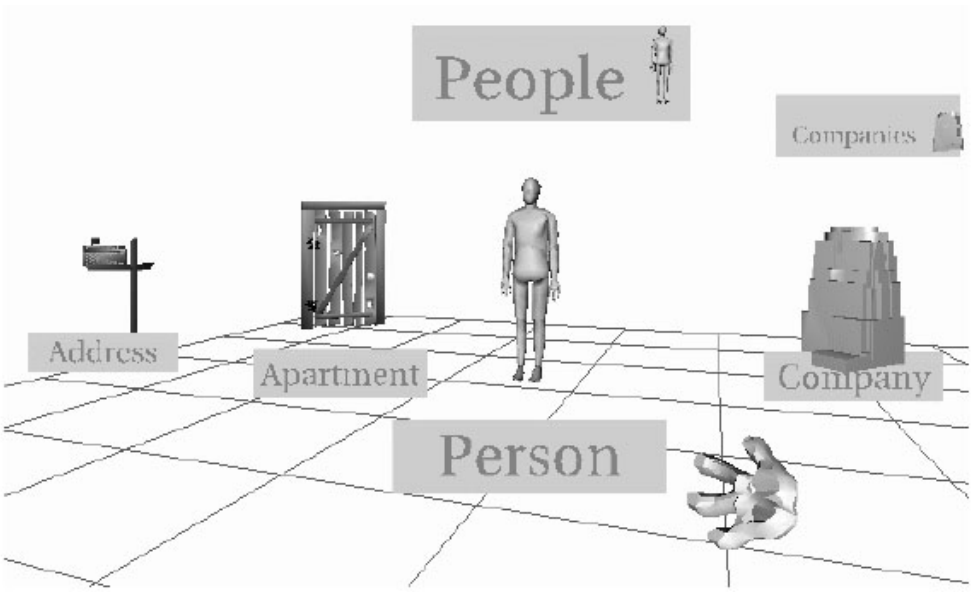

Figure 4. Representation of a database schema in Kaleidoscape

The system uses an interesting approach to deal with AND/OR connectors (another proposal can be found in [Jon99]). It is based on the flow model described previously by Schneiderman [Shn91]. We find it very useful and intuitive thus we have adopted it to our tool. A manner of work with Kaleidoscape is a little bit different than in VOODOO. The user freely "draws" a query rather than expands a graph of templates. Kaleidoquery is data flow-based which means that data "flows" from one place to another one. For instance – finding attribute values for names and ages of people means a "flow" from the People extent to the new results extent, but only selected attributes are able to pass the filter. Figure 5 shows an appropriate visual Kaleidoquery construct and Example 3 shows its OQL counterpart.

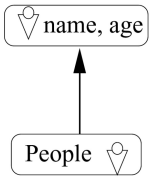

Figure 5. Kaleidoquery construct selecting name and age of the people.

```
select tuple(name:p.name, age:p.age)
from p in people
```

Example 3. OQL query selecting name and age of the people.

Another simple query selects all people whose age is less than 20 (see Figure 6 and Example 4).

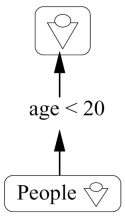

Figure 6. Kaleidoquery construct finding people with the age less than 20.

```
select p
from p in people
Where p.age < 20
```

Example 4. OQL statement finding people whose age is less than 20.

The above examples are quite intuitive, but when we try to formulate something more complicated, constructs become much less legible. For instance let's formulate a query finding all people working in companies located in England. Figure 7 and Example 5 contain appropriate queries.

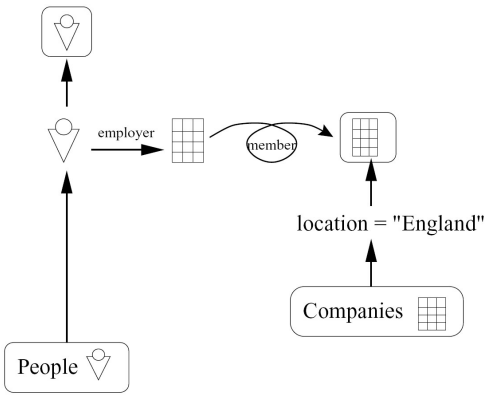

Figure 7. Kaleidoquery construct finding all people working in companies located in England.

```
select p
from p in people
where p.employer in
   (select c
    from c in Companies
    where c.location = „England”)
```

Example 5. OQL query finding all people working in companies located in England.

The last example present – intersection of two sets: all people who work for IBM and people whose an employer's location is London (see Figure 8 and Example 6). In Chapter 4.8 we will show how our metaphor deals with such queries.

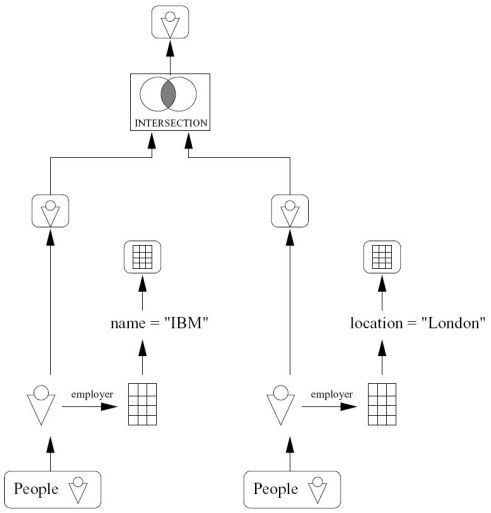

Figure 8. Kaleidoquery construct intersecting two sets: all people who work for IBM company and people whose employer's location is London

```
(select p
from p in people
where p.employer.name = 'IBM')
intersect
(select q
from q in people
where q.employer.location = 'London')
```

Example 6. OQL query intersecting two sets: all people who work for IBM company and people whose employer's location is London

Polaris [Sto02] (Figure 9), designed for relational databases, has some querying capabilities, but it seems that the major emphasis has been put on data visualization. Information retrieval is based on relational predicate filters working with particular fields, functions like min, max, etc.

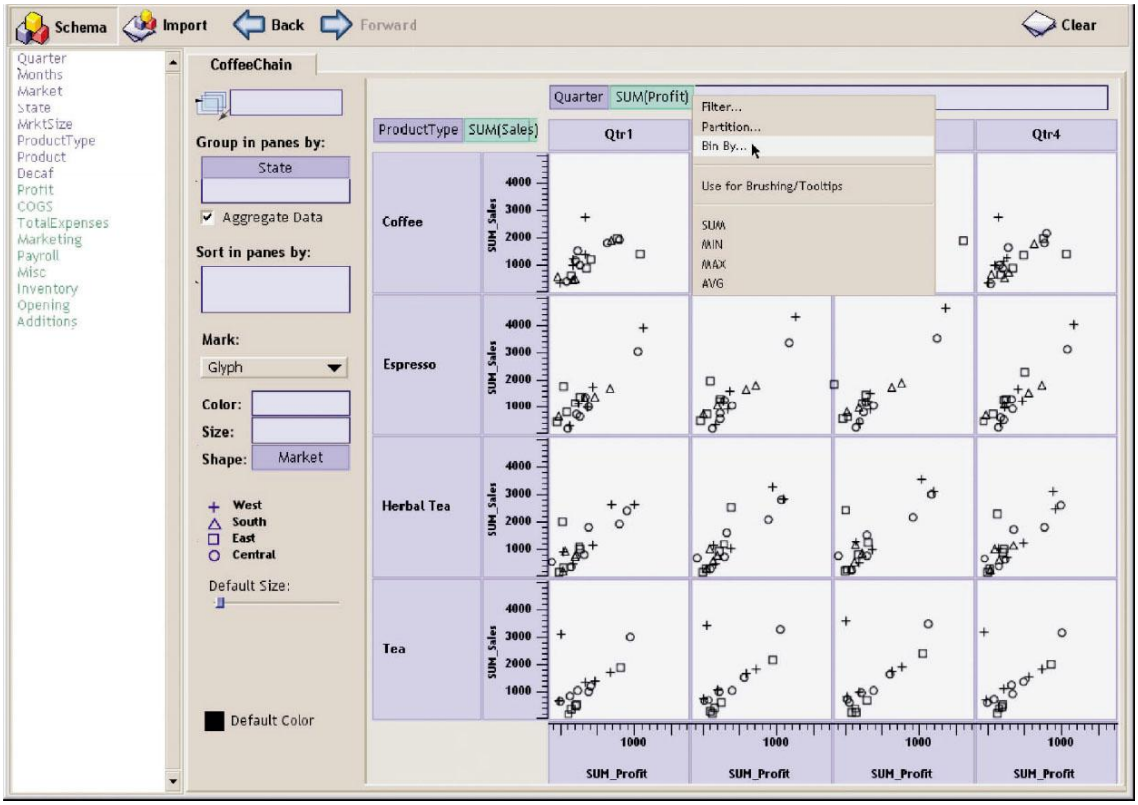

Figure 9. The Polaris user interface.

All visual query systems have some limitations. For instance, VOODOO allows the *group-by* construct to have *having* conditions for non *grouped-by* values and regular conditions for *grouped-by* values only. However, having in mind our target user (computer non-professional), we consider that the most important disadvantage of such systems is a manner of work. We believe that visual diagrams (for instance see Figure 3, Figure 7, Figure 8) containing queries are too complicated for naive users. Perhaps they would be useful, but only for computer literate users.

In contrast to visual querying that forces the user to draw a query and then to execute it, browsing systems allow querying a database in many steps. A typical example of a browsing system is GOOVI [Cas01] (Figure 10) developed by Cassel and Risch. Unfortunately, selecting of objects is done via a textual query editor. A strong point of the system is the ability to work with heterogeneous data sources (see chapter 2.3).

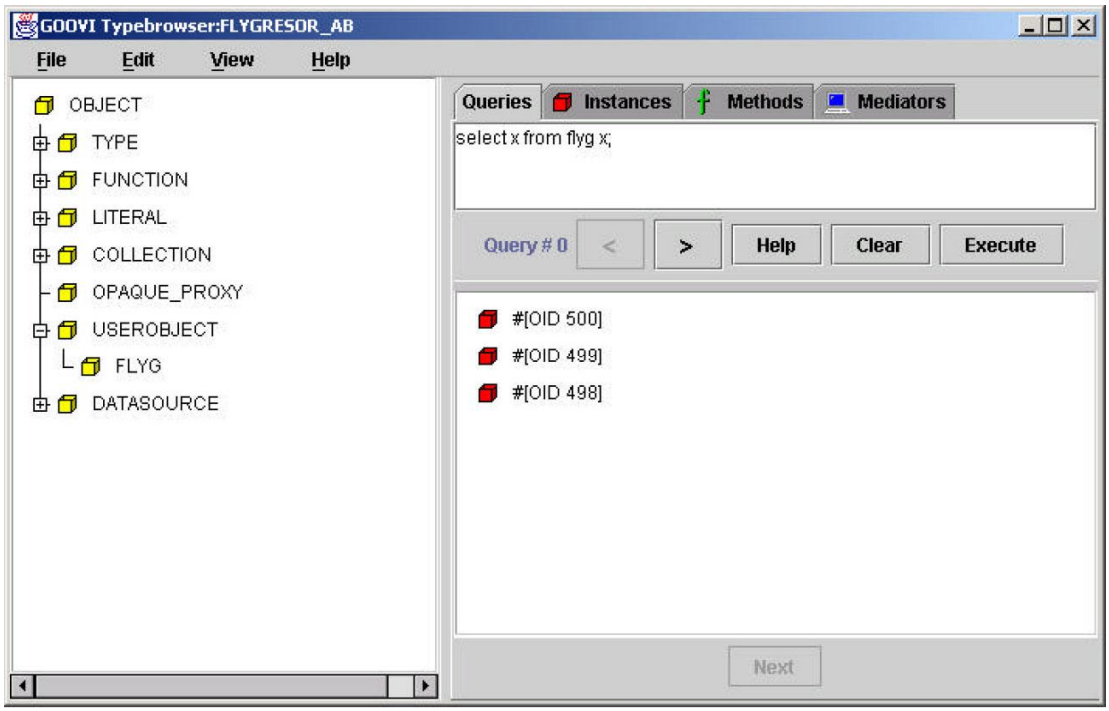

Figure 10. The GOOVI type browser.

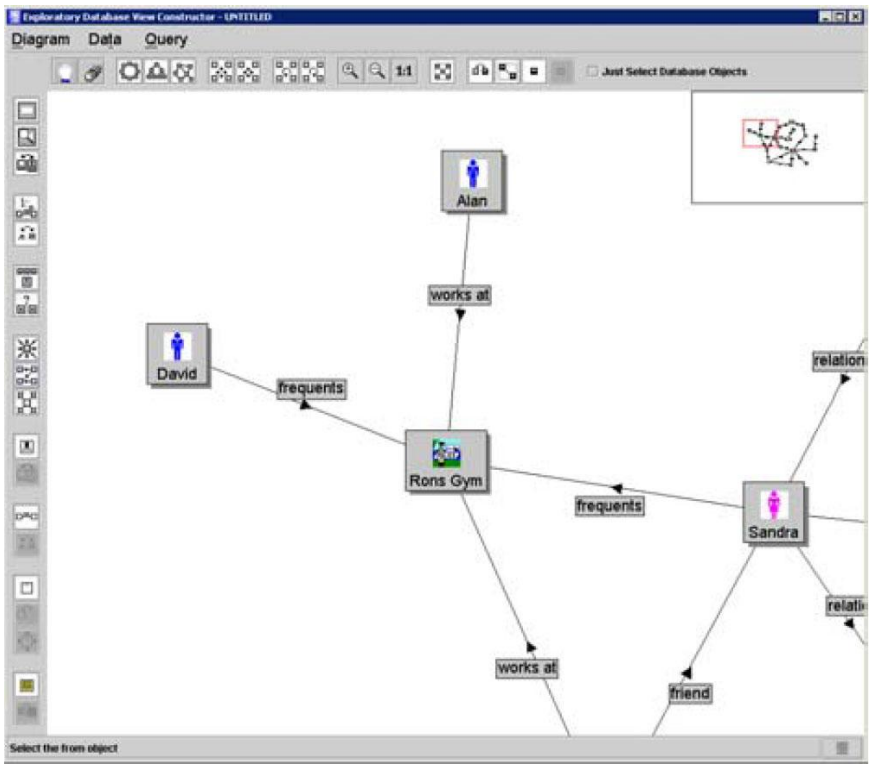

Figure 11. The Watson user interface.

Another interesting browser is Watson [Smi02] (Figure 11), 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. 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 a combination of these methods. The manner of work is similar to the metaphor that we have called extensional navigation (see chapter 3.2).

Browsing systems relay on manual navigation from one object to a next one. During browsing the user can read the content of selected objects. Browsing is the only option in situations when the user cannot define formally and precisely the criterion concerning the search goal.

2.2 Methods of Modifying Application's Functionalities

The most hard and universal method of modifying application's functionalities is generating a completely new system based on some existing framework. Depending on the designing/developing method, this approach is dedicated for particular groups of users (usually however not naive ones). DRIVE [Mit96a], [Mit96b] (Figure 12) 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. The specification of the interface is made in a textual language called NOODL. The model framework includes four main class categorizations, which are mapped to the language. Below we give a short description of each of them:

- The database component is responsible for communication with a database and contains a few classes. In the simplest case, a data class is reflection of the class from a database schema. Example 7 shows a simple definition of an artefact class for a virtual museum. Aside of two properties' definitions (name, catalog_id), contains reference to the `Artefact_Interface` class, which represents a GUI related task.

```
class Artefact
properties
    interface : Artefact_Interface ref referent
    name : Text
    catalog_id : Number
```

Example 7. Definition of a class representing a museum artefact in a NOODL database.

- The user component, which manages users' actions. Example 8 represents definition of some user class. The definition contains properties that specify kinds of task, which could be performed by the user. Component also has abstract classes with predefined user's rights models.

```
class User
properties
     accessors : #Interface ref users ;
     sophistication : Sophistication ref user ;
     authority : Authority ref user
operations
     task1 is self.task1a, self.task1b ;
     task1a is self.interface.intention1 ;
     task1b is self.interface.intention2
class Sophistication
property
     user : User ref sophistication
class Authority
property
     user : User ref authority
```

Example 8. Sample user class definition in NOODL.

- The interface component mainly defines appearance and actions related to the data. Example 9 presents a definition of the interface referenced in the Example 7. The intention of the interaction is described by `operation` and `intention` options. The interaction medium corresponds to a sequence of events in the definition. The effect of interaction is described by an interface class' trigger action. Notice that the visualization metaphor is defined by another class `Shape`. Separation of the interface, data and visualization gives a way to provide an alternative visualization mechanism.

```
class Artefact_Interface (* Interface Class*)
properties
     referent : Artefact ref interface ;
     metaphor : Shape ref interface ;
     museum : Museum_Interface ref artefacts (* composite *)
operation
     browse is self.metaphor.select
constraint
     self.metaphor.position.is_inside(self.museum.metaphor.extent)
trigger
     browse => self.museum.detail.referent(referent)
```

Example 9. Sample definition of an interface in NOODL.

Thanks to separation of data and interface, 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 kind of user (including casual ones) will accept it.

Teallach [Coo00], [Bar03], [Gri01] (Figure 13) employs the idea of a Model-Based User Interface Development Environment (MB-UIDE). It particularly supports specification of three models:

- domain,
- task,

- presentation.

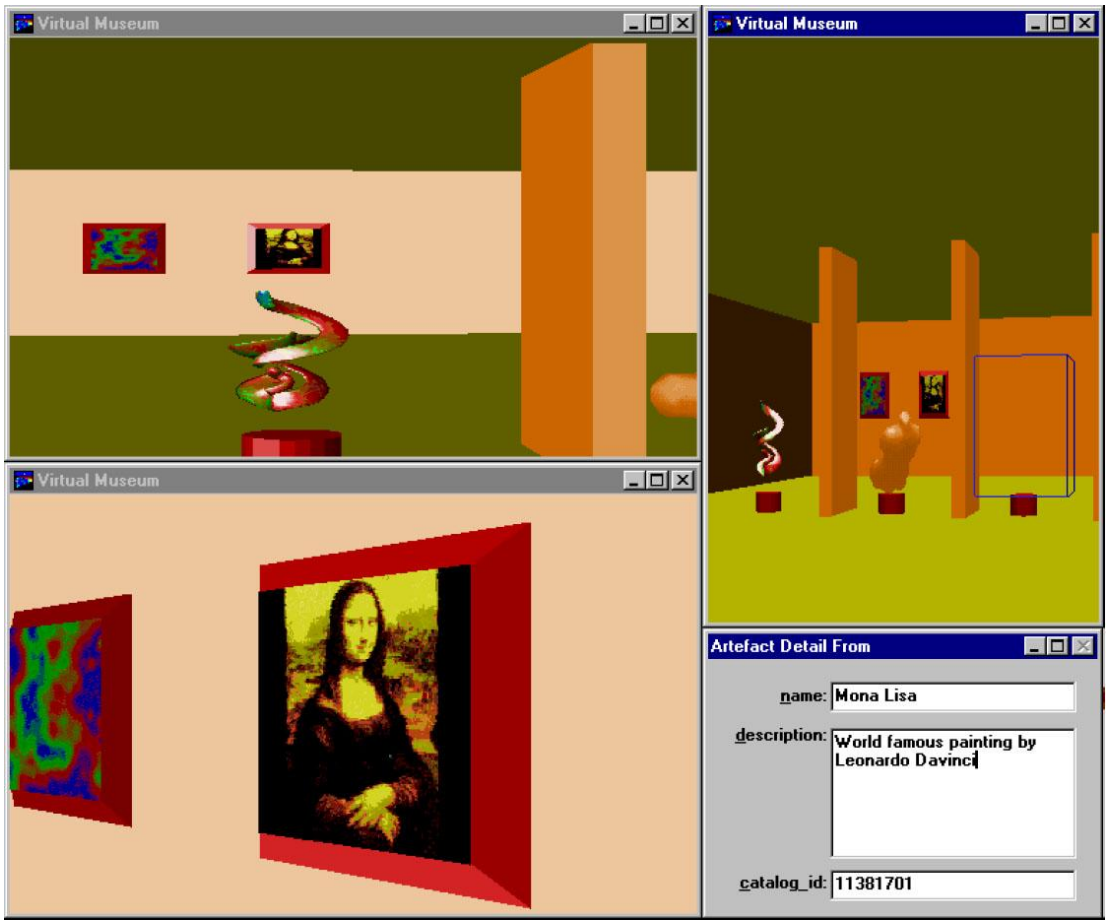

Figure 12. Virtual museum constructed in DRIVE.

Teallach domain model is based on the ODMG object database standard. Thanks to that solution, Teallach is able to automatically build internal domain model by extracting information from a database scheme. The model covers both application (transient) and database (persistent) objects using the same constructs. Its main role is to represent core application functionalities related to the database tasks such as queries, transactions, etc. Additionally the model is used to represent auxiliary data types used in runtime operation of the interface (i.e. class libraries used to manipulate data input to the system).

The Teallach task model is used to define the behaviour of the application – what a user can do with an interface. Also it plays an important role in cooperation between other models. The model is build as a goal-oriented task hierarchy, with leafs

representing action (automatic - i.e. updating of the form's date, manual – submitting of a form when completed) or interaction (i.e. obtaining a password's value from a user). Subtasks could be run on seven different ways, including sequential, optional and concurrent. Conditional or repeatable task are described in the UML's object constraint language (OCL). Each Teallach task is also capable to catch an exception and to be cancelled.

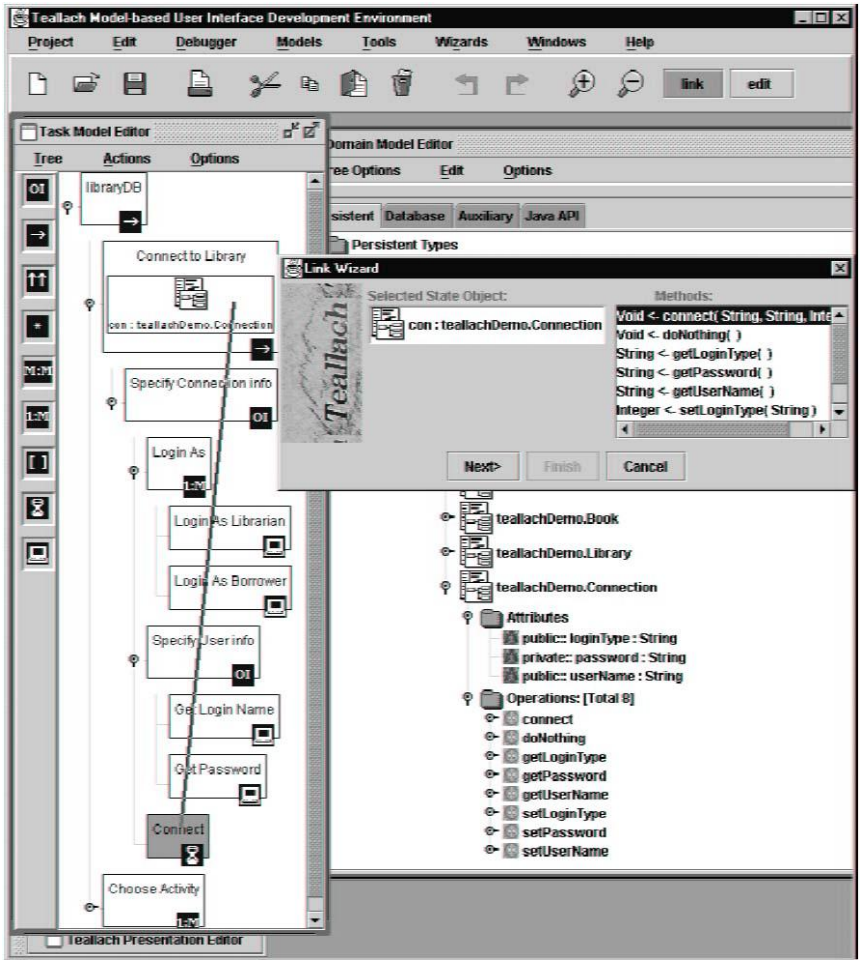

Figure 13. Teallach tool, showing a link between the task and domain models

The last model supported by Teallach, called presentation model, is used to define visual part of the user interface. The model contains two levels of abstraction:

- Abstract layer,
- Concrete layer.

The first one defines the interface in terms of high-level categories, without imposing particular interaction objects from the concrete layer. The layer contains definition of different fundamental kinds of component like: containers, inputters, displays, editors, choosers and action items. Each category is characterized by different set of operations.

The second layer simply contains user-interface widgets like: buttons, sliders, list boxes, combo boxes, panels, options, message boxes, etc. The described Teallach version uses the Java Swing set. However it is possible to implement custom components (in the JavaBeans technology).

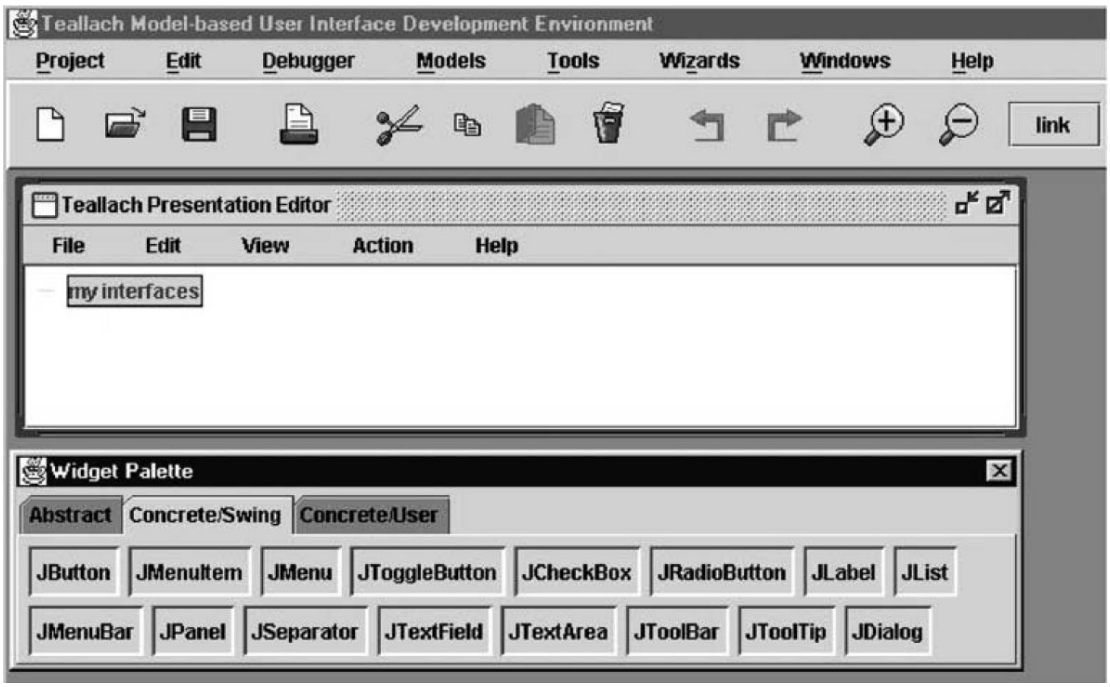

Figure 14. Teallach presentation model tool.

The whole process of creating the application is supported by specialized graphical editors (see Figure 14). The user builds an interface by linking together appropriate items from presentation and task models. To our knowledge, Teallach does not introduce any kind of built-in information retrieval capabilities. All 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 serious disadvantage, because there is no common and coherent basis of information retrieval methods.

It is common disadvantage for all the frameworks that developer has to start from a scratch (in terms of absence of some kind of existing application). These systems are really useful only if someone plans to create a completely new tool.

Visage [Rot97], [Der97] (Figure 15) is an example of another approach. The user interface itself contains some navigation methods for retrieval, which include:

- Navigating from an object to related objects,
- Aggregating a database object into a new one having selected properties from its elements.

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 dataset size that can be manipulated with no speed constraint. That is one of the reasons for using in Mavigator a fully-fledged programming language.

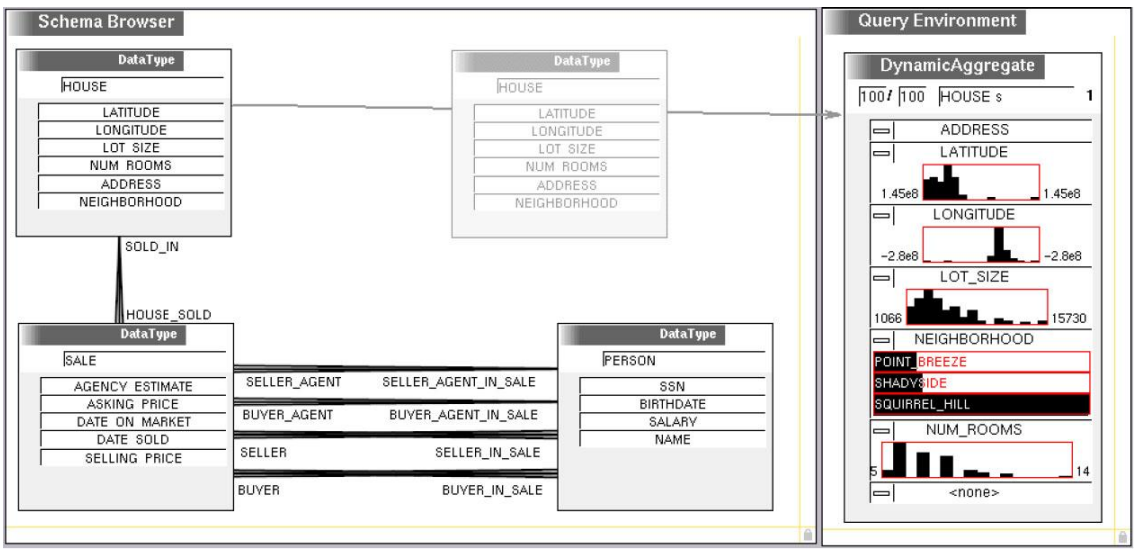

Figure 15. Database schema graph in the Visage

Visage uses the SageBrush tool [Rot94] to create map visualization. Figure 16 presents a screen from the editor during designing (left side) of the visualization (right side). Map shows location of the houses based on its addresses. Additionally different colors represent different neighbourhoods. Another interesting Visage's solution is

sliders' utilization (originally proposed by Ahlberg et al. in [Ahl92]), which is used to filter information. User can interactively drag sliders, to change minimum and maximum values of particular attribute. Only objects with attributes' values being inside the scope are selected.

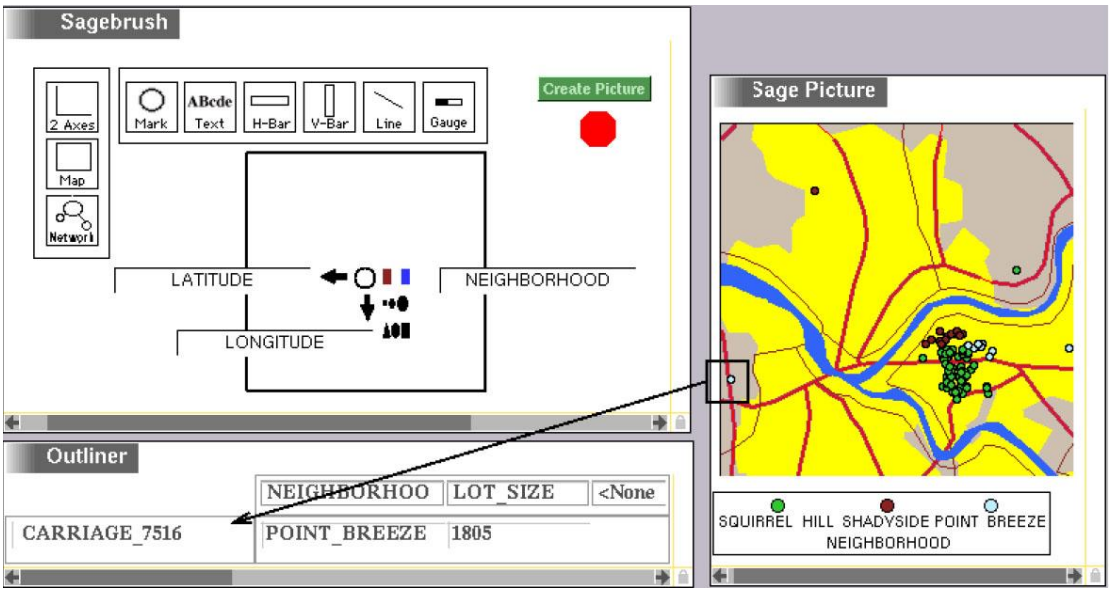

Figure 16. Constructing map visualization in the Visage.

2.3 Customizations of the Database Views

Database views are the subject of research and development for a very long time in the database community. However up to now, there are few proposals for object-oriented or XML-oriented environments, which are implemented, powerful and easy to use (for instance see [Koz03]). This is mainly caused by hard problems, including updates of virtual objects delivered by views. Our research concerns information retrieval capabilities, thus views' updating is beyond the scope of our investigations.

Non-updatable database views have a lot of applications including:

- hiding some parts of a schema, e.g., for security reasons,
- creating additional associations, which connect classes that are not physically connected,

- defining virtual associations among objects, by using a kind of a WHERE statement,
- hiding some intermediate classes – it is especially useful when one works with a relational data source where some many-to-many relationship is represented by two one-to-many tables and an intermediate table,
- changing some names, e.g. for some business reasons, for customization, or just for translating them into another language.

To our best knowledge Mavigator is a first system, which fully implements database views on the visual browser side, rather than in a database management system server. Usually graphical tools for information retrieval have a rich set of retrieval capabilities and just work with a particular data source; no user-side tuning of a tool through views is provided. On the other hand, systems dealing with views are just database servers and have no or very modest facilities for visual information retrieval.

One of the most advanced examples of views in visual tool is AMOS [Jos02] and its graphical browser GOOVI [Cas01] (Figure 10) developed by Cassel and Risch. This system uses mediators, which are counterparts to views. In contrast to Mavigator, the GOOVI browser is an independent tool, which only collaborates with AMOS. In other words, the browser has no views capabilities, but only works with the system, which supports them.

The AMOS system is a federation of independent mediator servers over a computer network. Each mediator is a separate database server with the ability to process object-oriented queries. Queries are formulated in the AmosQL language, which is similar to the object oriented part of SQL-99. A data source with querying capabilities uses AmosQL statements translated to its own query language. If a data source is not able to process queries, data are transparently imported into AMOS.

A frequent problem with different data sources is a common data model. The AMOS developers have decided to build an object-oriented extension of DAPLEX [Shi81]. AMOS covers various aspects of views: joining heterogeneous data source, retrieving/updating data and ability to work with functions/methods. The first property has been achieved by implementing special wrappers in programming languages such as

C, Java, and Lisp. The GOOVI browser is quite simple i.e. selecting objects is done via textual query editor.

The visual tool Watson and its information retrieval capabilities have been described in Chapter 2.1. Here we would like to mention that Watson's designers use the term *views*, however, not in the same context as we are. Comparing to our terms, their *view* is some state of the extensional navigation, i.e. some number of objects, which are connected by various links. Hence a Watson's *view* has nothing in common with a database schema.

3 Mavigator Metaphors

This chapter contains a detailed description of the ideas behind Mavigator's retrieval metaphors (see also [Trz04c]). At the beginning there is a general introduction. Then there are separate sub-chapters, with appropriate references, dedicated to each of them.

Mavigator is made up of five metaphors:

- Intensional navigation,
- Extensional navigation,
- Persistent baskets,
- Active extensions,
- Virtual schemas.

The subdivision of graphical querying to "intensional" and "extensional" can be found in [Bat91] and [Der97]. 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. Additionally, the user can use manual browsing via selected objects and manual selecting of objects through typed conditions similarly to an SQL *where* clause.

Intensional (Chapter 3.1) and extensional navigation (Chapter 3.2) 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. During both kind of navigation, objects being their results could be stored in a basket (Chapter 3.3). Because baskets are persistent, the objects could be used even in a next user's sessions. When the results of navigation (marked objects) are found (query results), user could perform some kind of actions defined using Active Extensions (Chapter 3.4), including projecting of the found objects (Chapter 3.4.2) or exporting them to another system (Chapter 3.4.3). All actions related

to the database schema graph, including both kind of navigation, take place in virtual schemas (Chapter 3.5), rather than in a physical database graph.

3.1 Intensional Navigation

Intensional navigation utilizes a database schema graph. Figure 17 shows a part of such a graph for the Northwind database, which is shipped with MS SQL Server. The graph consists of the following primitives:

- Vertices, which represent classes or collections of objects from the class. 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 UML terms),
- Labels with names of association roles. They define direction of the navigation for the particular association. This approach is similar to the one proposed in the ODMG standard, C++ binding.

The user can navigate through vertices via edges. Navigating means moving (in mind) from one vertex to another one. Navigation's process is connected with marking objects, which is our counterpart of getting query results. At the beginning of our research, we have been considering two approaches:

- The user can start navigating from any vertex and continue only through connected vertices,
- The user can navigate through connected vertices or simply can "jump" to completely unrelated vertices. After a "jump" there is a need to create a new starting set of marked objects (see further).

A potential disadvantage of the second approach is the chance that such a freedom in "jumping" will obscure the entire idea. However we believe that prohibiting such actions will definitely decrease the usefulness of the approach without clear improvements in easy-to-use. So we finally decide to employ the second approach, which gives more freedom to the user.

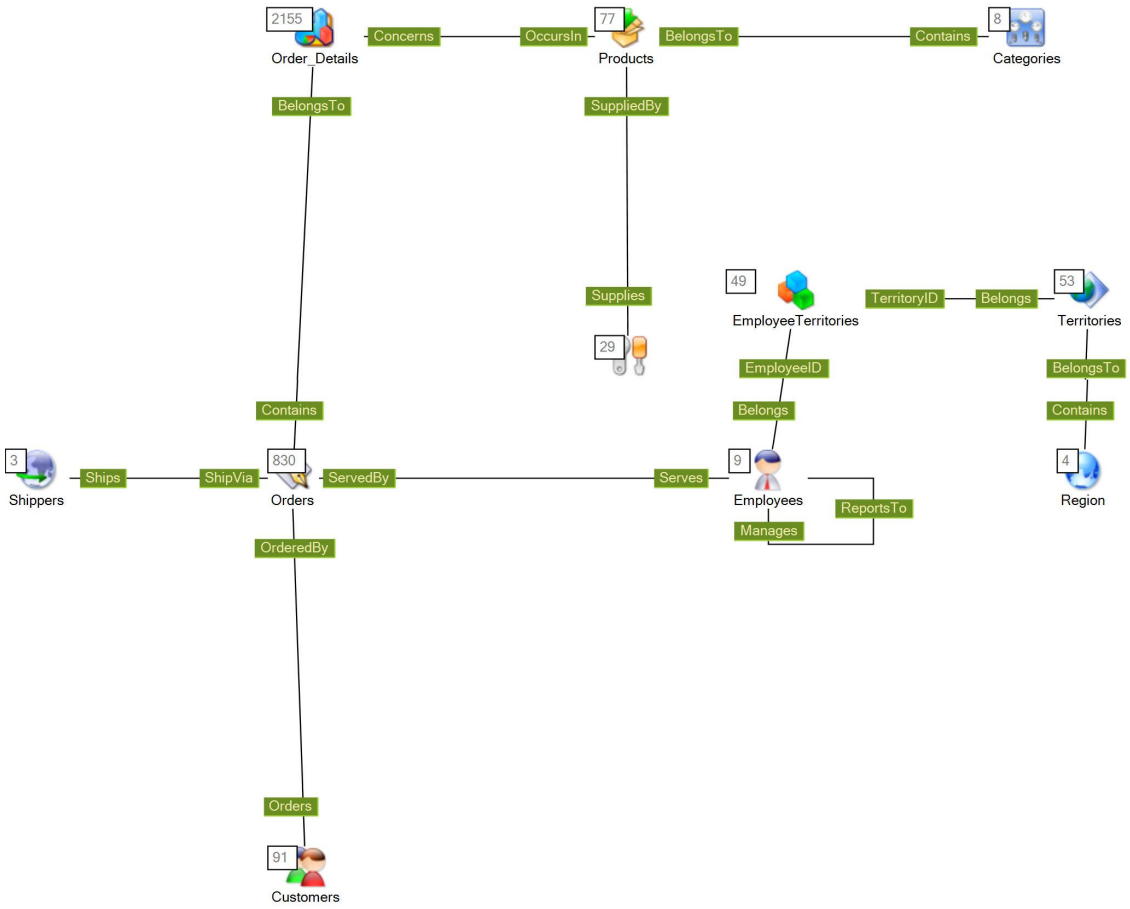

Figure 17. Intensional navigation graph

As we have mentioned previously, objects, which are relevant for the user (candidates to be within 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. Each of them is described below.

3.1.1 Filtering

Filtering is done through a predicate based on objects' attributes. The action causes marking those objects for which the corresponding predicate is true. There are two options: objects to mark are taken from a set of already marked objects or from the entire extension of the class. Filtering objects through a predicate is analogous to an SQL *where* clause. However, keeping in mind that our research concerns naïve users, we have to mention that some studies (for instance [Jon99]) show that naive users have problems with distinguishing AND and OR operators. This difficulty could have serious

impact on formulating right conditions. To deal with that problem, we have decided to employ graphical filter-flow approach proposed by Schneidermann [Shn91]. The metaphor uses the analogy to flowing liquid. Both operators perform the role of pipes, which the liquid flows through. Figure 18 explains the idea. Single predicates (A,…, E) are connected by lines-pipes, which define particular (AND, OR) dependencies. Bottom part of the figure contains textual definition of the predicate.

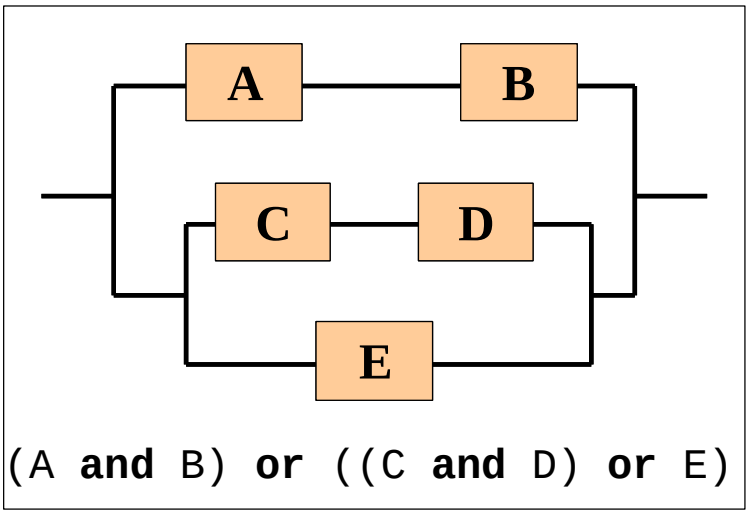

Figure 18. Illustration of the filter-flow approach to defining predicates.

Every single predicate contains statements like `Name = "Smith"` or `Salary < 5000` entered using user-friendly GUI (chapter 4.3.1).

3.1.2 Full Text Search

This activity marks all objects that contain all of the given keywords or phrases within any of their attributes. Similarly to filtering through a predicate, objects to be marked are taken from a set of already marked objects or from the entire extension of a class. There are also some possibilities to modify the algorithm. One of them is to use a Google-like algorithm, which gets objects not only with all entered keywords or phrases, but also takes into consideration the importance (or relevance) of particular words. Because research on such algorithms is beyond of the scope of this thesis, we plan to use some existing solution.

3.1.3 Manual Selection

Every object in our metaphor has some special attribute (called label), identifying it by comprehensive phrases. Thanks to that solution it is possible to show a list containing particular objects. Than user is able to mark or unmark particular objects manually. It is especially useful when the number of objects is not too large (i.e. when user already performed some filtering or navigating) and there are no common properties among them.

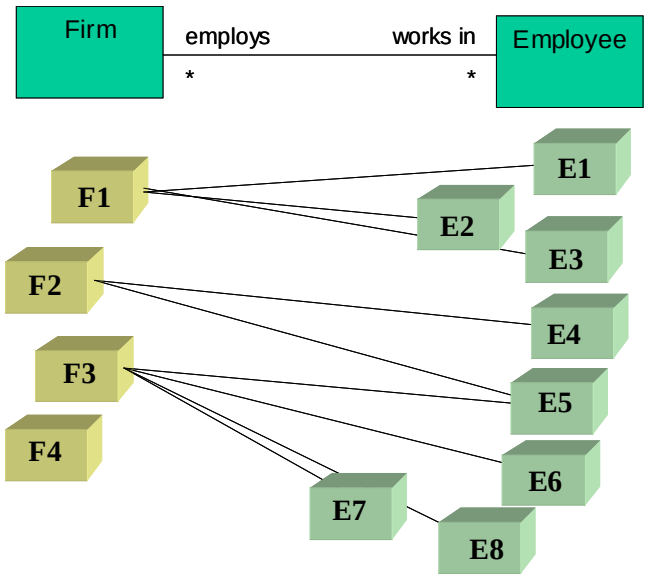

Figure 19. Marking objects via intensional navigation

3.1.4 Navigation

Navigation requires some starting point defined as a set of marking objects. The process means navigating from marked objects of one class, through a selected association role, to objects of another class. An object from a target class becomes marked if there is an association link to the object from a marked object in the source class. Figure 19 explains the idea. Let's assume that the *Firm* set of marked objects (i.e. previously filtered) has four marked objects: *F1,…,F4*. Than, navigating from *Firm* via *employs* cause marking eight objects: *E1,…,E8*. It is worth noting that:

- Object *F4* from the *firm* class is not connected with any objects from *Employee* class (firm does not employee people), which means that the result of navigation from this object is empty,

- Object *E5* from *Employee* class, is referenced two times: one from the object *F2* and one from the object *F3* (one person works in two firms). However the object will occur in the marked objects' set only once.

The concept of navigation is similar to path expressions in textual query languages.

A new set of marked objects, which is the result of navigation could:

- Replace the existing one (the one, which has existed in the target class before navigation has been performed; for the above examples there will be a set of marked objects associated with the *Employee* class),
- Be subtracted from the existing one,
- Be added (union) to the existing one,
- Be intersected with the existing one.

According to the above examples, two later options allow one to perform a query like: get all employees working for the "IBM" company **or**/**and** earning more than 5000 (if the existing set of marked objects contains employees earning more than 5000).

We believe that the options are easy to understand (even for naive users) and greatly enhance retrieval capabilities.

3.1.5 Basket Activities.

As we have mentioned previously, baskets are used to store search results and are seamlessly integrated with the marked objects concept. Its utilization in marking objects is supported by using the drag and drop metaphor. During intensional navigation, dragging and dropping the content of a basket on a class icon causes some operation on the marked objects of the class. New set of marked objects taken from the basket can:

- Replace the existing set of marked objects,
- Be summed with it,
- Be intersected with it,
- Be subtracted from it.

There is also a possibility to perform a reverse action. Dragging and dropping class's icon on the basket means creating a new sub-basket containing all the marked objects of the class. More information will be given in the chapter 3.3.

3.1.6 Marking Objects Using 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 a C# program the capability can also be used to mark objects.

3.1.7 Closing Remarks

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, browsing, manual selection 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. We think that adding them to the main metaphors could result in excessive complication of the approach. Instead of including them into intensional navigation, we propose to implement them as Active Extensions. Our naive user could ask a programmer to implement necessary additions in a way that he/she wants. Furthermore some of the aggregation functions, like sum or maximum, already have been implemented in our prototype. That approach has a big advantage – does not complicate metaphors and simultaneously allows using new solutions, if necessary.

3.2 Extensional Navigation

Extensional navigation takes place inside extensions of classes. Figure 20 shows a corresponding graph, which consists of:

- vertices, which represent particular objects, using same icon for group of objects belonging to the same class,
- edges indicating links,

- static labels with names of the associations' roles (not shown on Figure 20). They are hidden for the most time, because problems with legibility of the graph.

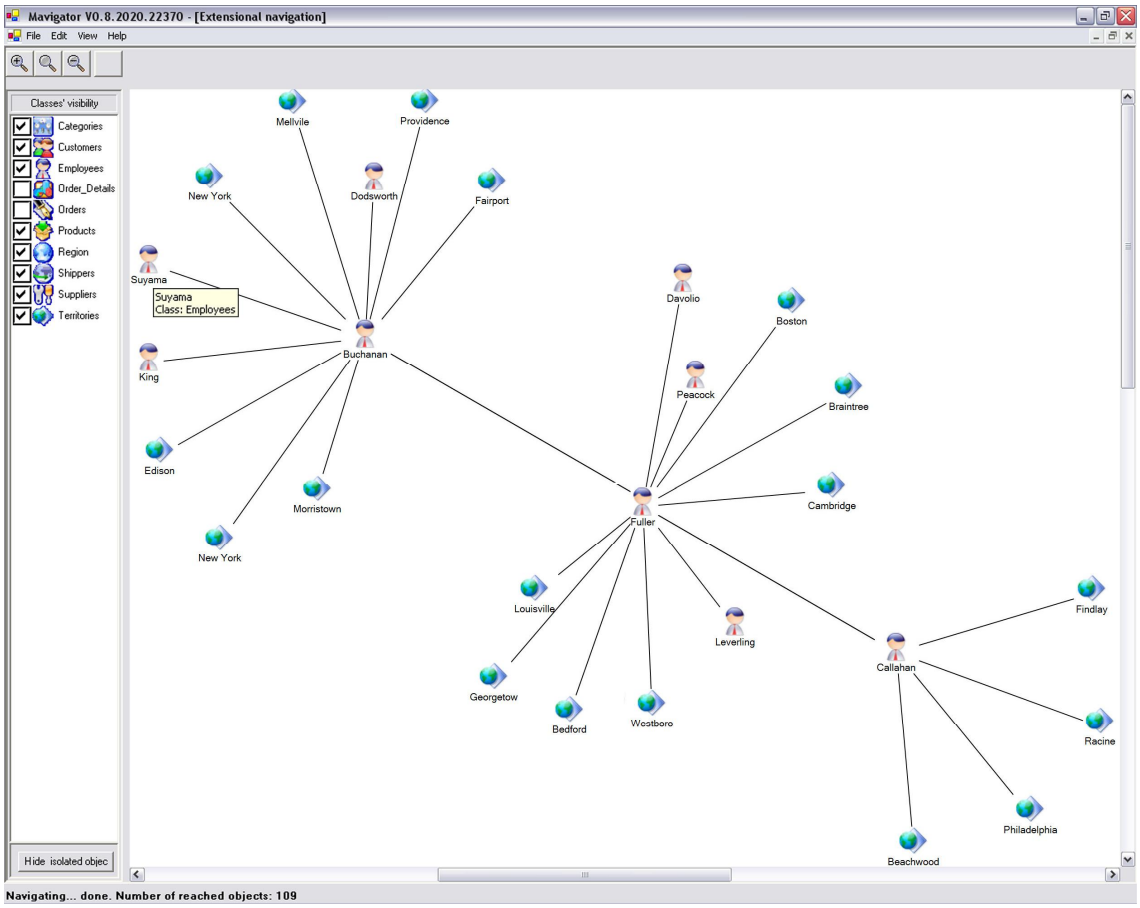

Figure 20. Extensional navigation graph

Extensional navigation, on contrary to the intensional one, is much more dynamic (graph "grows" during navigation). The user starts with one object and its neighbourhood. The last term defines objects and links connected with an "active" object. At the beginning, the "active" object is simply a starting object. During navigation, the active is changing to the currently selected one. When the user double clicks on a particular object, the object becomes active and an appropriate neighbourhood (objects and links) is downloaded from the database, which means "growing" of the graph. Thanks to the approach, the user can literally see the particular state of the data source.

There is a several ways of acquiring starting objects:

- Using baskets:
    - By dragging and dropping a basket or a particular object onto the extensional navigation surface,
    - By simply selecting an appropriate command from the object's menu,
- Choosing the option from the particular object's menu, when the list of objects is shown (see sub-chapter 3.1.3),
- During existing extensional navigation session, there is a way to start a new one with a particular object.

Extensional navigation is very 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. During the navigation session, there is a possibility to use a basket for storing temporary objects or to use them as starting points for the further navigation.

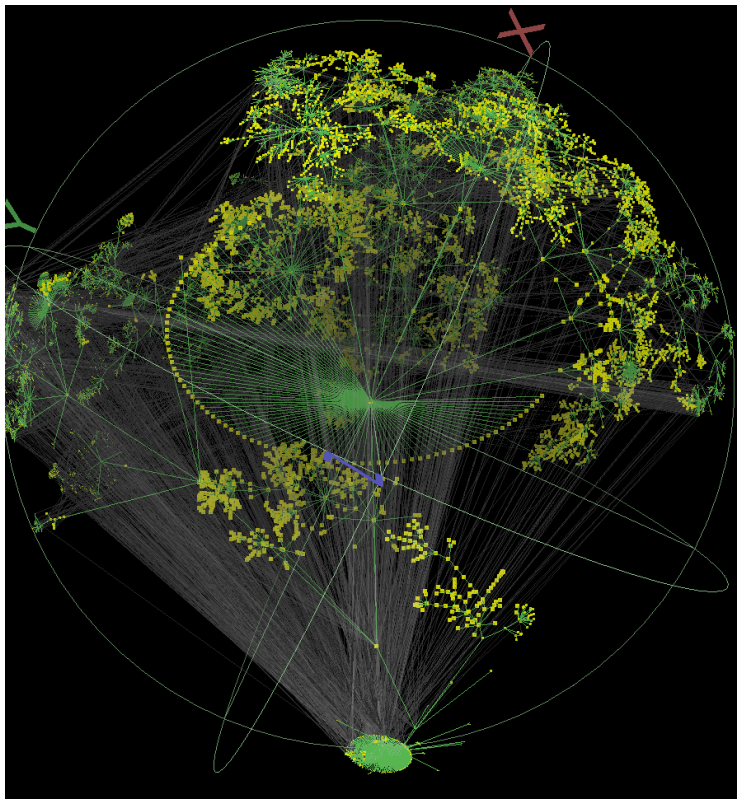

Figure 21. Screenshot generated by the Walrus - Graph Visualization Tool.

A serious issue related to the extensional navigation, is the legibility of a graph. There are a lot of approaches to visualize graphs, including advanced 3D techniques (an interesting example is [Wal], Figure 21). However, we had to find a solution, which will work in the iterative way. The reason of our decision is an intention to animate the entire process of layouting the graph. After some trials, we have decided to use a well-known algorithm based on the idea of balanced springs. The idea is that vertices, during performing layout, try to find the balanced localization; when they are too close to each other, they push away, when they are too far, they attract each other. Depending on some constants, there are possibilities to change the speed and distances. Additionally, there are some parts, which are responsible for avoiding dousing.

Another problem, which affects legibility of the graph, is the meaningful number of objects to visualize. There are two basic difficulties:

- Performance problems connected with processing and visualizing such a number of objects, which are especially important because the whole process is animated in real-time.
- Even if we use a good layouting algorithm, there might be problems with the placement of objects, caused by the fact that each object has its own graphical icon (which occupies some part of the surface) to show. That means that sometimes simply there will be no room on the surface to place an icon's object.

The solution to the first issue is related to implementation of the prototype therefore will be carefully described in Chapter 5.

The answer to the second problem is more complicated, because the solution is multiplane and concerns mainly a user interface, thus will be described in Chapter 4. However, the basic assumptions are the following:

- A user is able to filter objects to show, which means hiding/showing objects from particular classes,
- There will be an automatic way to hide isolated objects, i.e. those ones, which are not connected to other objects,

- A user can manually reconstruct the graph, by dragging particular nodes, which causes smooth re-layouting,
- It will be possible to pin some objects to the surface excluding them from the automatic repositioning.

It is worth noting that the entire extensional navigation, similarly like the intensional one, takes place inside a virtual schema, rather than inside a physical one. This means that the user sees not real links from the database, but virtual ones defined by the creator of a particular virtual schema. This feature could be useful in the context of legibility of a graph too. It is possible to define such a virtual schema, which will not contain unnecessary classes or links; therefore they will not be obscuring the visualization.

3.3 Baskets

A search for information requires a place to store the search results. It is especially true if searching is operated in an *ad hoc* manner, when the user switches or even jumps from an object to an object. This kind of searching principally corresponds to extensional navigation, but even during intensional navigation, there is a way to "jump" from one class to another. To satisfy the user's needs for storing objects (or groups of objects) we have introduced the basket metaphor.

Baskets are persistent storages of search results. They store two kinds of entities:

- "Objects"; as a matter of fact baskets store unique object identifiers (OIDs) rather then physical objects. This approach has significant consequence: if an object is unreachable in the database (i.e. because of security reasons), it also unreachable from the basket. However such an object will not be removed automatically from the basket. It is possible to remove it manually. Instead, special information to the user will be shown. This is mainly caused by the fact that unreachable objects could be reachable again after some time.
- Sub-baskets. Every sub-basket can contain other sub-baskets or objects. The number of nesting levels is unlimited.

Figure 22 shows a sample visualization of the basket containing three sub-baskets and five objects. As it can be seen, 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 OIDs, because a special string represents the objects visually. The strings are taken automatically by a dedicated method having OID as a parameter and returning the string from the database. Additionally every object could have attached a special note, which can store any kind of information (i.e. the reason for adding it to the 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. There is no limitation concerning the number of baskets or the number of objects stored in a basket.

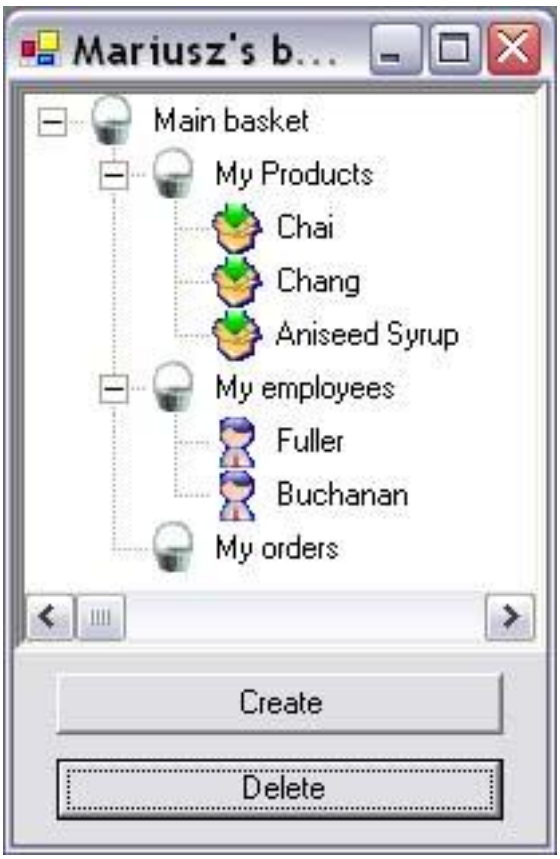

Figure 22. Visualization of the basket containing three sub-baskets and five objects

The metaphor is defined in an intuitive way and allows coherent cooperation between different methods of information retrieval. Below we give detailed description of all basket activities.

3.3.1 Creating a New Basket

The user is able to create a new basket at any time during the session. The system records a couple of properties for each basket including time of the creation, owner of the basket and description.

3.3.2 Changing Basket Properties

Each basket has own set of properties. Some of them like name and description can be changed. However, there are some properties i.e. time of basket's creation, which cannot be changed by the user.

3.3.3 Removing Selected Items

The basket's metaphor allows one to remove both kinds of basket's items. Depending on the kind of item, the semantic of the process is a bit different. In particular:

- Removing an object means eliminating it from the basket, without deleting from the data source,
- Removing a sub-basket means removing all items in the sub-basket: its sub-baskets and its objects.

3.3.4 Performing Operations on Two Baskets

Before an operation, each of two baskets being operation's argument is virtually processed. All objects from particular basket argument and its sub-baskets are put on the list, which means that the structure of sub-baskets is ignored during operations. There is also a possibility to allow or disallow for repetitions for OIDs. Default setting prohibits repetitions of the OIDs. Below we have enumerated operations, which could be performed on two baskets:

- Sum of baskets,
- Intersection of baskets,
- Set-theoretic difference of baskets.

The metaphor allows storing the user action result in a newly created basket or replacing the content of one of the baskets participating in the operation.

3.3.5 Using in Extensional Navigation

Basket metaphor is also heavily utilized during the extensional navigation. There are two kinds of actions:

- Drag an object from the basket and drop it onto an extensional navigation surface. As the result, the neighbourhood (other objects and links) of a dropped object will be downloaded from the repository.
- Drag particular object from the extensional navigation surface and drop it onto particular sub-basket. The action cause storing the object in the chosen sub-basket. There is also a possibility to annotate dropped object by attaching any kind of text. Annotations are not shared among different users.

3.3.6 Using in Intensional Navigation

Similarly to extensional navigation, the intensional navigation also utilizes the basket metaphor. Extensively, there are two possibilities:

- Dragging a basket and dropping it onto class's visualization in the intensional navigation surface. The operation acts as follows:
  - Objects from all sub-baskets, belonging to the particular class, are put on the list (which means that only objects belonging to the particular class are processed, the rest is ignored). During adding to the list, multiple instances of the same objects are prohibited.
  - There are four possible operations, between objects from the list and existing set of marked objects: replacing, adding, intersecting, subtracting.
- Dragging a class's icon and dropping it on the basket area. A new basket containing all marked objects from the dragged class will be created.

Baskets allow storing selected OIDs in a very intuitive and structured way. Navigation can be stopped at any time and temporary results (currently selected/marked objects) can be named and stored for future use.

3.4 Active Extensions

In our opinion, naive users do not want exaggerated number of functions. Such a typical user wants an application with functions, which will be used in everyday work, without functionalities, which will be used very rarely. The problem is with defining the scope of the functionality, because for the same kind of applications (information retrieval tools) different users will need different functions. We believe that the solution could be based on two assumptions:

- Supply the users with some basic retrieval metaphors and functions,
- Provide a way to extend existing functionalities, based on particular user's needs.

All previously described Mavigator's metaphors (see chapters 3.1 - 3.3) are solutions to the first assumption and Active Extensions are our answer to user's needs concerning modification of the application functionality.

At the beginning, when we started to think about methods of extending existing functionalities, two different approaches come to our minds:

- Utilizing some kind of a graphical metaphor like in [Mit96a], [Mit96b] and [Coo00], [Bar03]. They are tradeoffs between the power and easy-to-use. They were designed to be easy enough for the target user. However, we think that our target user could not be able to use such complex metaphors. 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, however, that a Mavigator's user is not a programmer and will not be able to use such extensions. Hence some professional programmer must come into play.

As we mentioned earlier, Mavigator already employees some information retrieval metaphors (see chapter 3.1 - 3.3), 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 (which is

important for an Active Extensions programmer – see chapter 5), but guarantees sufficient flexibility. Our solution requires collaboration of an end user with a programmer who will write the code accomplishing the required functionalities. Despite the assumption that Active Extensions have been designed for modifying a query result, it is possible to program any kind of tools. This is caused by the fact that each active extension's program has access to entire data source and a list of processed objects.

When we have decided that our approach should employ a programming language, one more thing should be defined. Thinking about a programming language there are two general possibilities:

- use a special, probably scripted one, created especially for the purposes of the system,
- use already existing one.

The first approach has advantages like:

- possibility of creating such constructs, which help in creating particular solutions,
- opportunity of bringing a language to much a higher abstraction's level than existing one.

The first advantage, which is very important (because facilitate the work), could be realized in the second approach too (we will show it in chapter 5.6). The second benefit would be very important if the target user of the application would create extensions. Though, we believe, our target user will not be interested in (or simply will not be able to do) creating such extensions. And for the regular programmer, who will make extensions, the abstraction level of the regular programming languages is high enough. Another important factor could be performance – typical, script languages are quite slow. Important premise is also significant extra effort needed to design and implement a new programming (even scripted) language. Thus, based on the above arguments, we have decided to use, a regular existing programming language.

The current prototype, which has been developed for purposes of this thesis, uses Microsoft C# as a language for active extensions. A programmer is aware of the Mavigator meta-data environment (for implementation details see chapter 5), which

allows him/her to write a source code of the required functionality in C#. Writing of the Active Extension source code is done in the 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 the query result (see chapter 3.1, 3.2) or objects recorded in a user basket (see chapter 3.3). The functionality of such programs is unlimited and its potential utilizations include:

- statistical analyses,
- data-mining,
- cooperation (exporting data) with other systems,
- implementing completely another approaches to information retrieval,
- extending existing mechanism to information retrieval by using concepts like data aggregation,
- generating reports.

Next three sub-chapters present its particular applications, which has been implemented and evaluated in our prototype.

3.4.1 Simple Active Extensions

One of the simplest types of Active Extensions could be those ones, which count something. In the current Mavigator's prototype we have implemented most popular aggregate functions:

- minimal attribute's value,
- maximal attribute's value,
- average attribute's value.

All of them are very easy in use. All that the users have to do, after selecting a particular type of calculation, is to select (in the query result) a particular attribute. After that the result is shown to the user.

3.4.2 Active Projections

More sophisticated example of the Active Extensions are Active projections (Figure 23), which allow visualizing a set of objects where the position (in terms of x, y coordinates) of each of them is based on value of particular objects' attributes. Current implementation uses two axes (2D), which allow visualizing dependencies of two attributes. Figure 23 shows objects of class product and theirs dependencies between sample attributes:

- unit's price,
- units in stock.

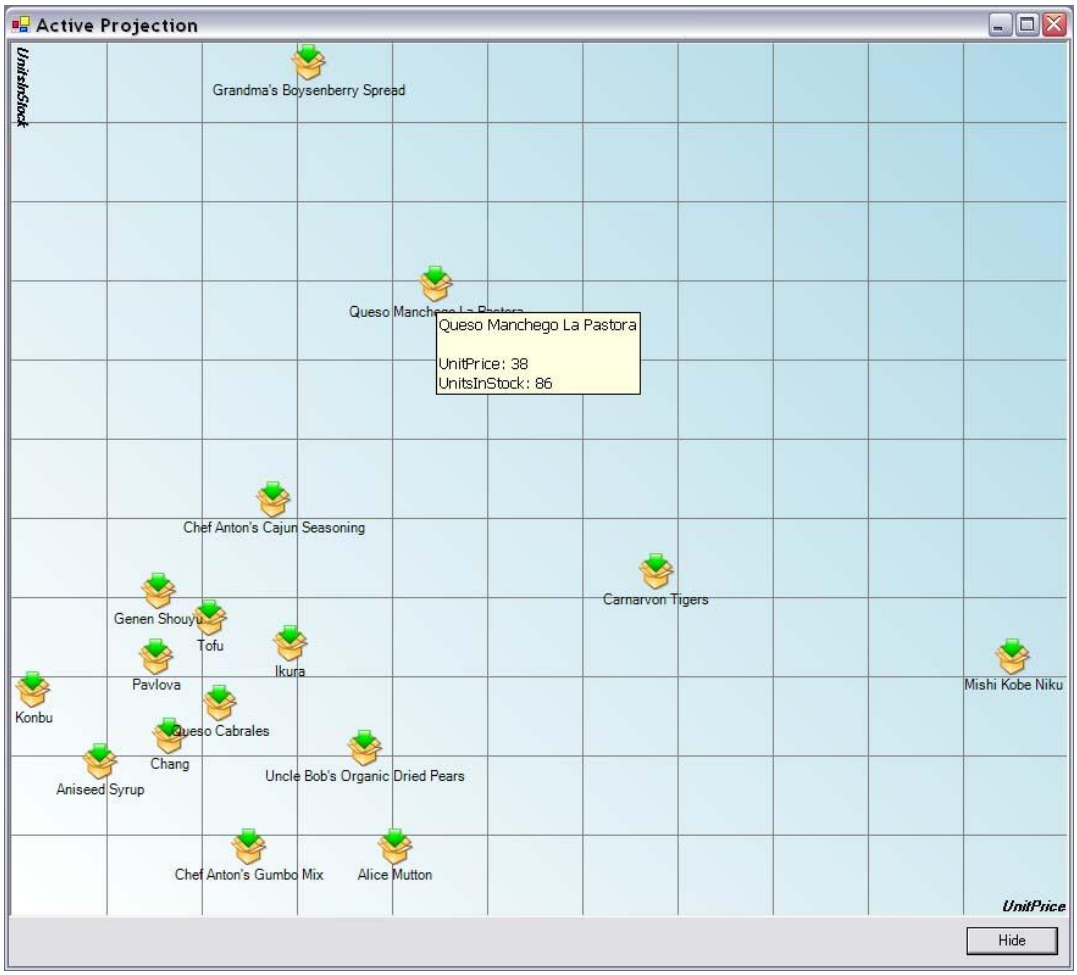

Figure 23. Active projections.

Active Projections make it possible to identify some groups of objects. For instance it is easy to see (Figure 23) that we have a little amount of cheap products and (generally) more cheap products than expensive ones.

We take into consideration increasing number of attributes without adding another dimension. In particular [Sto02], [Ber83], [Now98] propose to use the features of visual icons such as:

- shape,
- size,
- orientation,
- color,
- texture

and correlate them with values of additional attributes. For instance, the size of a product icon could be proportional to its price.

Besides the visual analysis of objects dependencies it is also possible to utilize projections in more active fashion. There are two options:

- 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).

We also consider a third type of active behaviour. It would be some kind of extensional navigation (see chapter 3.2) on projection's surface: clicking on object's visualization causes visualizing its neighbourhood. However we are not sure if such an approach would not lead to misunderstanding. The reason is that some objects (the projected ones), will have right placement (based on attributes' values), but not all objects from the neighbourhood could be appropriately placed. This is caused by the fact that some of them (or even most of them) will come from a class, which attributes are not projected on the surface. A partial solution could be some kind of visual differentiation, for instance by painting a special icon or border.

3.4.3 Objects Exporters

Introducing new application requires creating a way to exchange data with other, existing systems. Of course one approach to the problem could be based on implementing filters to some popular tools. However there is always a possibility that a user would like to cooperate with an application, which is not available through the filters. Thus we have another utilization of Active Extensions. Objects exporters allow cooperation with other software systems. Having a query result (or basket's content), 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 or browsing. The current prototype exports to XML files, which could be post-processed by a number of modern applications.

3.5 Virtual Schemas

All Mavigator's metaphors heavily depend on a database schema graph. Hence, it would be useful, if there will be a way to customize a particular schema for a particular user. Normally, such requirements are satisfied by database views. However, there are barriers in using them. Some of the database management systems do not provide views functionality or block their usage for particular user. In our approach, we have been created some kind of database views on the application side. The corresponding module is called Virtual Schemas.

Each Mavigator's data source has at least one virtual schema, which is a direct reflection of the database data. As will be shown in Chapter 5.2.5, a single virtual schema has very low resource demand, which means that Mavigator is capable to work with a big number of virtual schemas. Every operation on virtual schemas is executed exactly the same way, as it would be on real data. Thus, the virtuality of objects, according to the database views principles is perfectly transparent for the user.

Generally, Virtual Schemas allow creating customized database views, which consist of the following elements:

- Virtual associations, which reflect any dependency among classes. The definition of an association is made up of a few items:
    - two role names (direct and reverse),

    - two class names,
    - two query language statements, which define a target set of objects. In the simplest (default) case queries return all objects from the reverse classes. However, such queries can contain any valid query language constructs.
- Virtual attributes, which describe objects' properties. Each attribute has:
    - A name, which is shown to the user. Such a name is completely independent from the physical name of the class's attribute.
    - A query language statement, which defines this attribute. In the simplest case it is just the name of a physical attribute. More advanced utilization could use procedures, constants, etc. Thanks to that solution, it is even possible to create completely new attributes, which do not have counterparts in the physical database schema.
- Classes, which are counterparts of the physical collections from the database. We have decided to use only classes, which already exist in the database. Such an approach does not require materializing objects by the view and ensures that stored (in a basket) object will be correctly recreated even in a different virtual schema (if its class will be accessible). Each class is defined by:
    - Name shown to the user (independent from the class physical name),
    - Name used to access the class in the database (one of the physical names from the database).

Description of the virtual schema is stored in an XML file. Most of the above items are defined using Stack-Based Query Language (SBQL). For more information implementation details see chapter 5.2.5.

Figure 24 shows an example of the virtual schema. All of the examples are based on the well-known Northwind relational database, which is shipped with MS SQL

Server. Comparing to the originally relational schema, only names of the association roles has been changed. Thanks to that diagram is much more readable.

Following sub-chapters show particular applications of Virtual schemas.

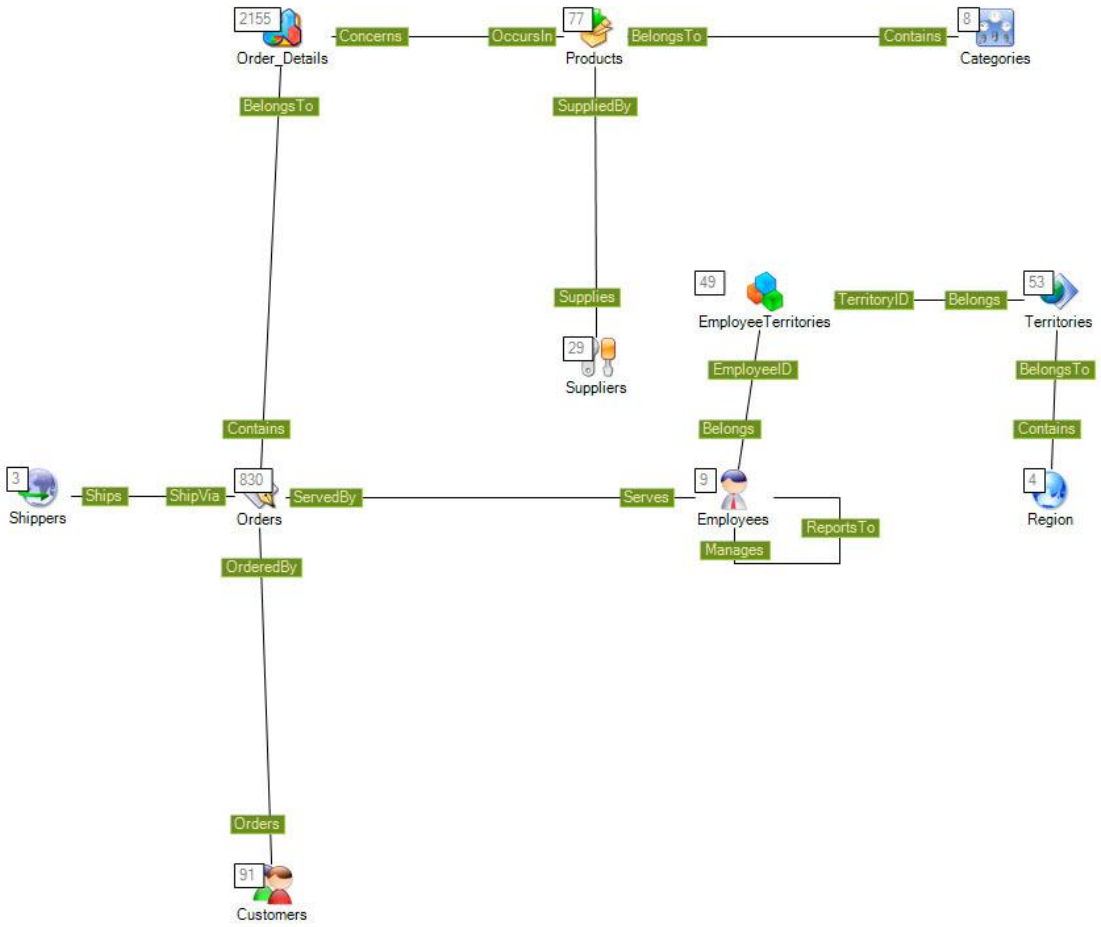

Figure 24. Example of a database virtual schema

3.5.1 Determining or Changing Names of Associations' Roles

Sometimes there might be a need to change names of association roles. That case proceeds in case of Northwind database. Because the original Northwind schema is based on a relational data source, it utilizes relationships rather than associations. Relationships are identified by primary/foreign keys, and have no names. Such dependencies can be mapped as associations having proper names on their ends. For instance, having `CustomerID`, we can define the association named *OrderedBy* between *Orders* and *Customers*.

Another reason to changing names of the association could be caused by the need of translating the system into another language. By using a dedicated virtual schema, it could be achieved easily and efficiently.

3.5.2 Creating New Connections between Classes

Creating new connections between classes does not introduce any new information to the system. However it could be very useful for simplifying user's work. For instance, referring to the Northwind database, we would like to know which companies supply products for a particular order. Instead of navigating through two classes (*OrderDetails*, *Products*) we create a new association between *Orders* and *Suppliers*. It is quite easy to achieve using path expression of the query language, which defines an essential part of the virtual association. Example 10 shows appropriate code (for navigation in two directions) in SBQL.

```
Orders.Concerns.Order_Details.ProductID.Products.SupplierID.Suppliers;
Suppliers.Supplies.Products.Occurs.Order_Details.OrderID.Orders;
```

Example 10. Partial definition of the association between two classes.

3.5.3 More Accurate Specifying Objects from the Target Class

Default behaviour of the typical association is returning all objects accessible from another object. Sometimes it might be useful to refine in details which objects are interesting for the user. Let's assume that we would like to analyze only recent information stored in the database. Then we create an appropriate virtual schema. Among other information we want to know only recent orders served by particular employee. In that case, we extend the definition of the Serves role (between classes *Employees* and *Orders*) with particular `WHERE` clause, which will check date of the order and returns only objects (actually ids of the objects) with the current date. Example 11 shows an appropriate code – part (a) contains general definition and part (b) presents association, which returns only up-to-date objects of the orders class.

```
(a)          Employees.ServedBy.Orders;

(b)          Employees.ServedBy.Orders WHERE OrderDate = NOW();
```

Example 11. Definitions of the two associations: (a) returns all objects, (b) returns only up-to-date objects.

3.5.4 Hiding some Classes

It could be useful because of some security reason or just to simplify user's schema. All we have to do is removing particular classes' definitions from the virtual schema. Of course, names of the removed classes cannot be used in other definition (i.e. in virtual associations).

3.5.5 Hiding Intermediate Classes

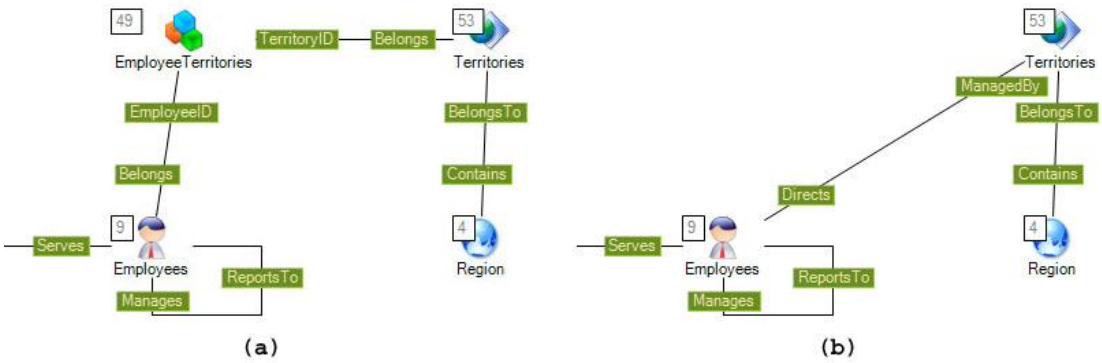

Figure 25. Illustration of hiding intermediary classes.

In a relational schema we are not able to present dependencies (relations) between classes (tables) with many-to-many relationships. In such a case we have to introduce an intermediate class (table). In an object-oriented model those classes could be connected directly, without introducing any extra classes. Hence, working with a relational data source or even data imported from the relational database, could be easier if those classes would be hidden. This situation happens in the case of Northwind database. Figure 25 shows two parts of graph from the Figure 24. Class *EmployeeTerritories* is a typical intermediate class (with no attributes), which is necessary only to illustrate many-to-many relationship between *Employees* and

Terrotories. Part (a) of the Figure 25 shows the originally graph; part (b) shows the appropriate part without the mentioned class, which improves legibility. This kind of modification requires only a few changes to the virtual schema definition:

- hiding class *EmployeeTerritories* (removing from the definition of the virtual schema),

```
(a)     Employees.Belongs.EmployeeTerritories;
        EmployeeTerritories.EmployeeID.Employees;

(b)     EmployeeTerritories.TerritoryID.Territories;
        Territories.Belongs.EmployeeTerritories;
```

Example 12. Partial definitions of two associations, utilizing intermediate class.

- modifying two path expressions describing an association. Example 12 (a, b) shows parts of two definitions for two associations connecting “business” classes and intermediary class. Example 13 shows partial definition for new association (the one, which connects “business” classes without intermediate one).

```
Employees.Belongs.EmployeeTerritories.TerritoryID.Territories;
Territories.Belongs.EmployeeTerritories.EmployeeID.Employees;
```

Example 13. Partial definition of the association, which “hides” intermediate class.

4 User Interface for Mavigator

This chapter is devoted to a user interface of the Mavigator prototype. The first sub-chapter gives background information defining a target user and requirements. The second one places Mavigator as a Windows application and tells how to start the work. Next sub-chapters discuss particular solutions dedicated to the metaphors. Chapter 4.7 describes supporting techniques common for all items of the GUI. The last chapter shows real-life case studies based on the Northwind database.

4.1 Background

The Mavigator prototype implements all Mavigator's metaphors described in Chapter 3. Hence, it enables the user to perform a search for information in a connected data source. Then, information, which has been found, could be processed by any of the Active Extensions, including Active Projections or exporting to another computer system.

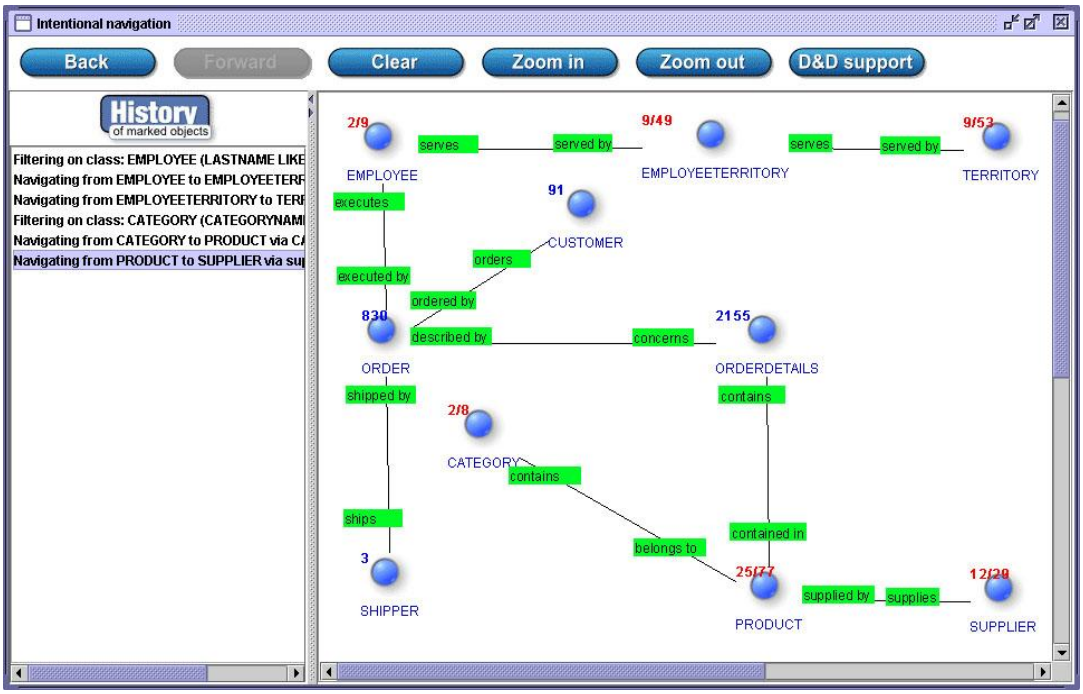

Figure 26. Structural Knowledge Graph Navigator (SKGN) during intensional navigation

Except Mavigator, there is another implemented prototype, which follows the Mavigator metaphor. It is Mavigator's predecessor called Structural Knowledge Graph Navigator (SKGN), developed during the European project ICONS. Figure 26 shows SKGN during working with Structural Fund Project Knowledge Portal database. However SKGN, as a first prototype, implements only three of five metaphors:

- intensional navigation,
- extensional navigation,
- baskets.

Because of the limitations, SKGN is not described in this dissertation. The information could be found in [Trz04a], [Trz04b].

4.1.1 Target User

It is pretty obvious that a user interface must comply with the user kind. For instance, advanced system administrators usually prefer textual interfaces (console-based) rather then graphical ones. On the other hand, inexperienced users choose interfaces based on graphical widgets. As we have mentioned in Chapter 1, our user has been specified as a computer non-professional. To be more precise, we would like to define him/her as a person, who meets the following assumptions:

- trained in using some popular programs such as MS Word or Excel,
- educated but not necessary in the computer science or even science. For instance it could be a data (statistical) analyst, broker, lawyer or dealer,
- highly motivated. This property is always required in order to learn some new metaphors and techniques.

We believe that a person meeting above requirements will be able to understand and efficiently utilize our metaphors.

Besides a target user, who deals with information retrieval, there is also another user (or users), who will be responsible for:

- designing and implementing Active Extensions. This activity requires a regular programmer – in case of the current prototype it should be a person, who is able to write programs in Microsoft C#.

- creating Virtual Schemas. This activity requires a bit different knowledge that the above one. A person responsible for the task of creating (or at least changing) virtual schemas must be able to write queries in database query language. Hence it could be a database administrator or (sometimes) appropriately trained programmer.

4.1.2 Requirements

Taking into considerations all the assumptions defined in the previous sub-chapter, the only sensible interface for our target user will be the graphical one. Such an interface should be distinguished by following three general properties:

- ability to search for information, which must be accomplished in an easy and intuitive way,
- capability to store information, which has been found. Those capabilities must apply not only at the end of the entire process, but also to any intermediate results.
- possibility of subsequent processing of information, which has been found. Those possibilities include, but are not limited to:
  - running additional functions,
  - performing various analysis,
  - exporting data to other computer systems.
- modifying of the system allowing:
  - adding new functionalities,
  - changing the data source schema graph, which is an essential part of all the information retrieval metaphors.

Besides requirements, which arise from the application kind, there are some general user interface properties, common for all proper interfaces. The prime rule is that a user must have feeling that he/she is in charge and run the system and not just the opposite. It is especially important if the system is dedicated to the inexperienced user. Such a user often will not be able to distinguish if he/she “did something wrong” or the

system works improperly. To achieve that, the entire system design must be conformed to some general principles including:

- All actions taken by the user must have some response from the system, for instance: if a user chose some option from the menu, which starts some action, at the end there must be some information to the user. Depending on the action kind (or action importance), the response could be shown in the modal dialog box or just in the status bar.
- Avoiding "freezing" the system as a result long-term action. Starting an action, which could run for a long time (i.e. processing query result or connecting to the server), must be connected with showing special widget. The widget should at least show "working" of the system or, where it is possible, progress of the processing. It is also possible to warn a user that chosen action may take a long time.
- During the whole time of working with the system, the application should support user awareness. The term means that the user is aware of his/her actions. It could be achieved by using special techniques like status bar or tool tips. Also there are some metaphors, which are especially useful to support user's awareness. For instance, baskets allow storing objects during the whole process of information retrieval, which means that a user is able to go back to his/her data even after changing the way of working with the application.

It is worth noting that fulfilment of the above principles means some extra effort for the analysts, designers and programmers of the system. Some of them cause serious changes to the system's architecture. For instance, in most cases implementing progress bar requires introducing threads to the application, which is always complicated. However, in real application, we should not sacrifice proper user interface to the easy of development.

4.2 General information

Mavigator has been implemented as the MDI (Multi Document Interface) application in Microsoft C#, which is a part of the Microsoft .NET platform. MDI application has a big advantage over SDI (Single Document Interface): it is possible to

have many windows to be opened. It is especially useful in case of Mavigator, which coherently groups a couple of metaphors. We can deal with a couple of interacting metaphors (i.e. intensional navigation and baskets) at the same time or we can switch from one metaphor to another one (i.e. from intensional to extensional).

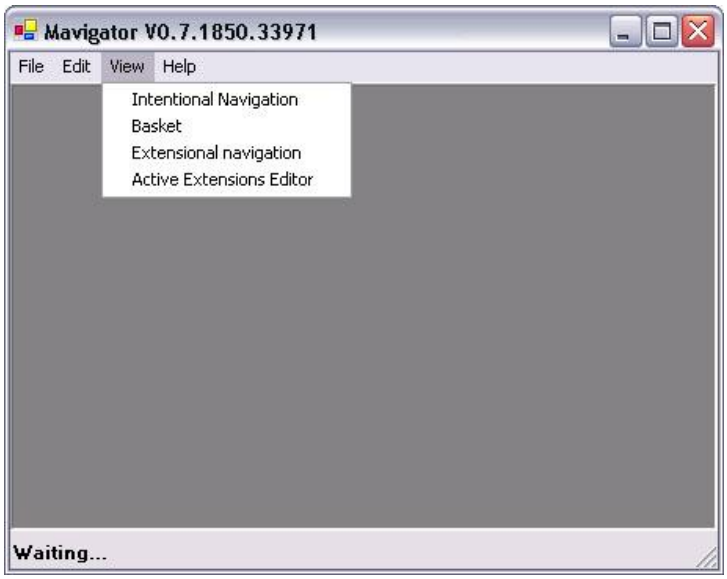

Figure 27. Mavigator prototype just after starting.

Figure 27 shows the application just after starting. Mavigator conforms to the standard Window's application rules:

- at the top of the window there is a menu,
- at the bottom there is a status bar, where are displayed messages,
- at the right-top corner there are standard buttons.

All main actions are started from the menu – menu for the View is shown on the mentioned figure. After choosing particular option, appropriate window is shown to the user. Each window has its own set of buttons placed on the toolbar at the top of the window. Such assumptions facilitate learning process especially for the users, who know applications like MS Word or MS Excel.

The first thing, which should be done after running the application for the first time, is creating a new user. Depending on a data source kind, the process could have a different course, for instance some data source does not require password. Figure 28

shows window for an implemented wrapper to the ODRA prototype database. In this case the user has to write a database address, a database name, a user name, a password (two times) and a particular virtual schema. The check box also must be selected. Opening ordinary connection (after creating a new user) is almost the same: the only differences are that a password is written once and check box is not selected.

Figure 28. Open database window.

If everything will go fine, the application is connected to the data source and user can start working with the data.

4.3 Intensional Navigation

Intensional navigation has been designed as a primary way for information retrieval. Detailed description of the metaphor has been given in Chapter 3.1. In this chapter we will recall only the general assumptions, which will facilitate understanding this chapter. Figure 29 shows a Mavigator's window during an intensional navigation session.

The basic assumptions are as follows:

- Objects, which are important for the user (the result of the query) are marked, which means added to the set of marked objects.
- Each class has its own set of marked objects.
- There is a couple of marking techniques:

- filtering (see Chapter 4.3.1),
- navigating (see Chapter 4.3.2),
- manual operations (see Chapter 4.3.4),
- basket activities (see Chapter 4.5.3).

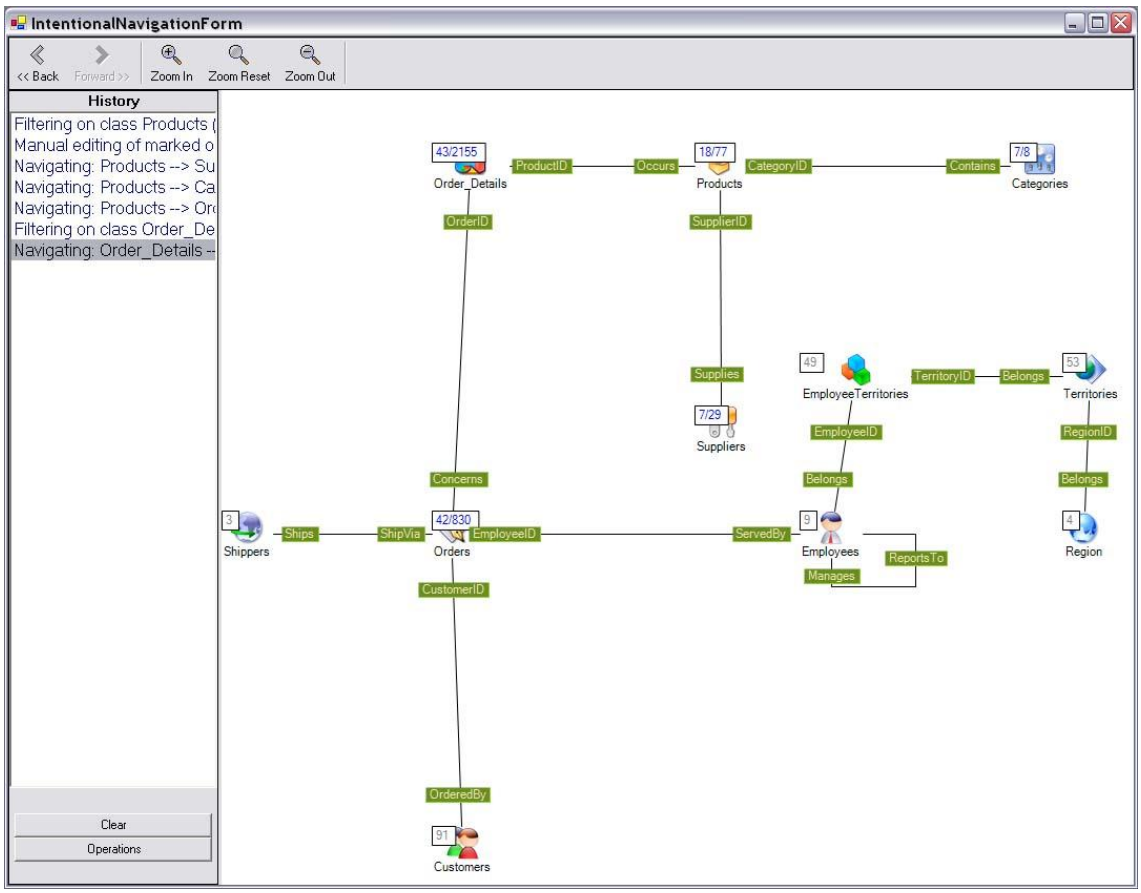

Figure 29. Mavigator's window during Intensional navigation session.

- During the whole process, there is a possibility to use baskets as a store for marked objects.
- Each set of marked objects could be processed by any of the Active Extensions.
- All user's actions are performed transparently on a chosen Virtual Schema rather than on a physical data itself.

4.3.1 Filtering

In order to filter objects from a particular class, the user has to click with right mouse button on the desired class and choose appropriate option from the menu. Figure 30 shows suitable menu for the Products class.

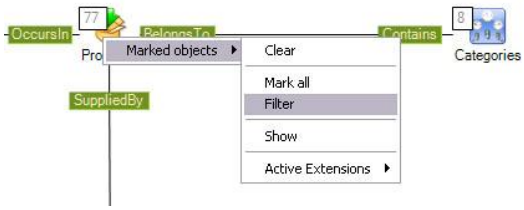

Figure 30. Choosing filtering marked objects from the context menu.

Filtering of the objects is based on the criterion formulated by the user. Because surveys [Shn91] show that computer non-professionals have problems with distinguishing AND/OR, the whole process follows a flow model and is performed visually.

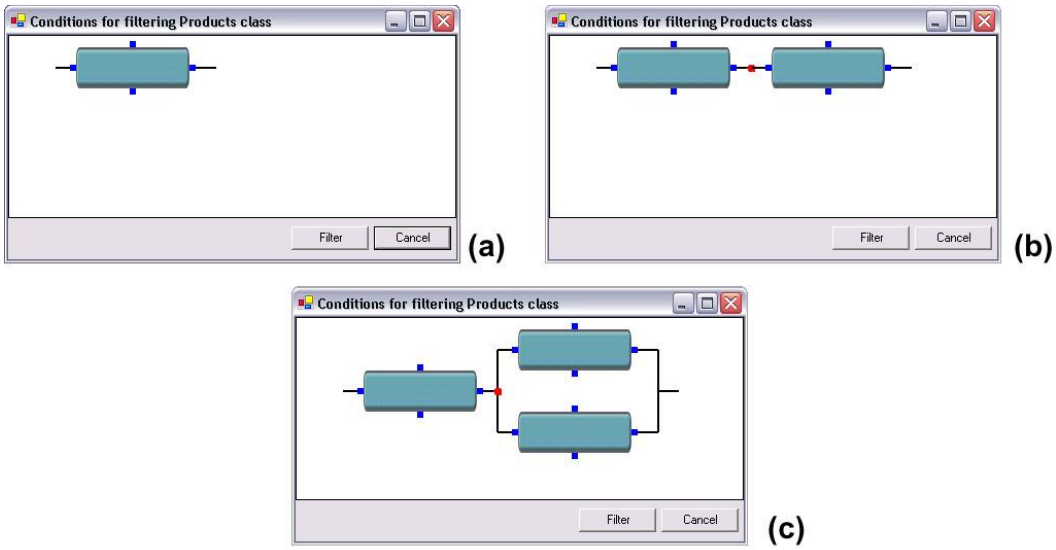

Figure 31. Visual formulating of a filtering criterion.

A user starts with an empty, single predicate icon visualized as a blue, rounded rectangle (part (a) of the Figure 31). Next empty predicates are added after clicking appropriate button. Left and right buttons (small, light blue dots) add predicates in

“horizontal” directions, which means “AND”. Top and bottom buttons add predicates in “vertical” directions, which symbolizes “OR”.

Let’s assume that user would like to formulate filtering criterion like those from Example 14 – give me all products where product name starts with M and number of items (units) is more than 12 or unit’s price is less then 40.

```
ProductName = 'M*' AND (UnitsInStock > 12 OR UnitPrice < 40)
```

Example 14. Sample criterion for filtering (textual version)

As mentioned previously, a user starts with a single, empty predicate icon (part (a) of the Figure 31). Then after clicking right “growing” button, a next predicate icon has been added (part (b) of the Figure 31). Finally user click bottom button of the second predicate, which cause adding a last predicate icon. The result is shown on the part (c) of the Figure 31. Now user should “fill” empty predicates with appropriate conditions. This process is supported by another window shown on Figure 32. All user have to do is choose (from the combo box) attribute, operator and type value. There are also tool tips for each item describing possible operations.

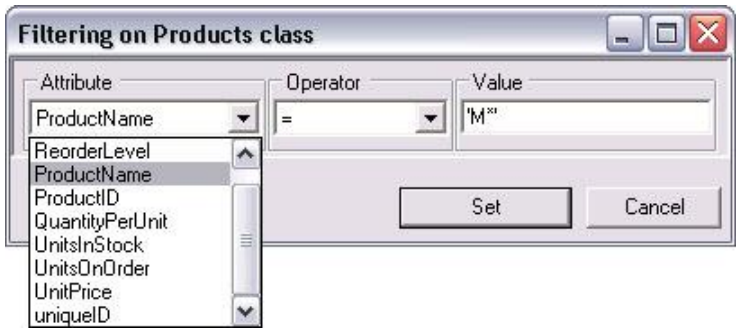

Figure 32. Formulating a single predicate

The final criterion (after formulating all predicates), which is a visual counterpart of the Example 14, shows Figure 33.

The process of formulating criterion is not completed without ways for structure (and, or relations) modifications. This could be achieved by using drag and drop techniques. The user can drag a predicate and drop it on:

- another predicate icon, which means replacement,
- a small, red dot, which means moving to the “AND” position.

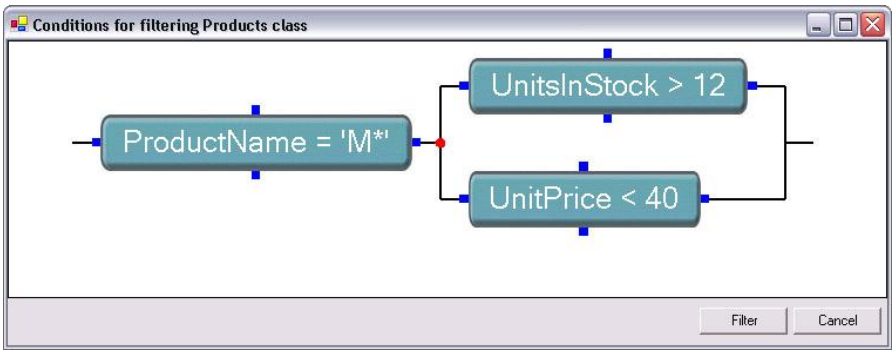

Figure 33. Visual counterpart of the criterion from Example 14

It is also possible to click on previously formulated predicate and change its contents. In this case, the user sees a window with appropriates widgets containing previously entered data, which could be changed.

When the user finishes formulating criterion and clicks the *Filter* button, the following action is done:

- The system starts validating the predicates (its completeness) without sending for evaluation;
- If predicates are complete (all parts are defined), then the criterion are send to the data source for evaluating; otherwise the user sees an error message;
- Objects, which conform to the criterion, are marked (added to the set of marked objects for particular class).

The whole process is very easy (even for computer non-professionals) and intuitive. The constructed criterion could contain unlimited number of predicates, connected in any way.

There is also a possibility to mark all objects from the class or clear existing set of marked objects (appropriate options from the context menu shown on Figure 30).

4.3.2 Navigating

Navigating is a frequently utilized technique, thus it should be especially well (from two points of view: ergonomic and performance) designed. Briefly speaking (for detailed description see Chapter 3.1.4): it is based on moving from one set of marked object, via selected association role, to the target class. As a result, target class has a new set of market objects.

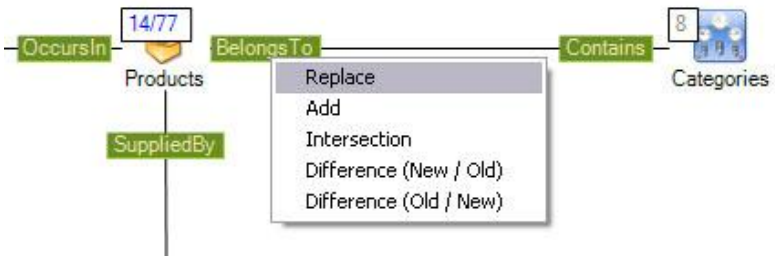

Figure 34. Using navigating to marking objects.

In terms of the user interface, the user has to:

- Choose any class as a starting point. In case if there will be no marked objects, some of them must be marked. Otherwise the result of navigation will be an empty set.
- Decide which association role will be used and then click on it.
- Select navigation policy in the context menu (Figure 34). The following options could be selected:
    - Replace means that the new set of marked objects replace the previous one,
    - Add will perform sum of two sets: the new one and the previous one, without repetitions,
    - Intersections will select objects existing in both sets,
    - The last two options allow performing difference on two sets, replacing two arguments accordingly.

During the navigation (communicating with the data source), progress bar is shown. After successful operation, selected association and target class flashes for a while.

4.3.3 History of Marking Objects

Due to the flexible metaphor, Mavigator's user is able to coherently utilize many different ways for marking objects. This is of course a big advantage, which gives total freedom in using the system. On the other hand, it could lead to some kind of disorder (decreasing of the user awareness), because of the many actions, which the user is able to perform. To avoid this, all user's actions concerning marking objects are recorded on a special list.

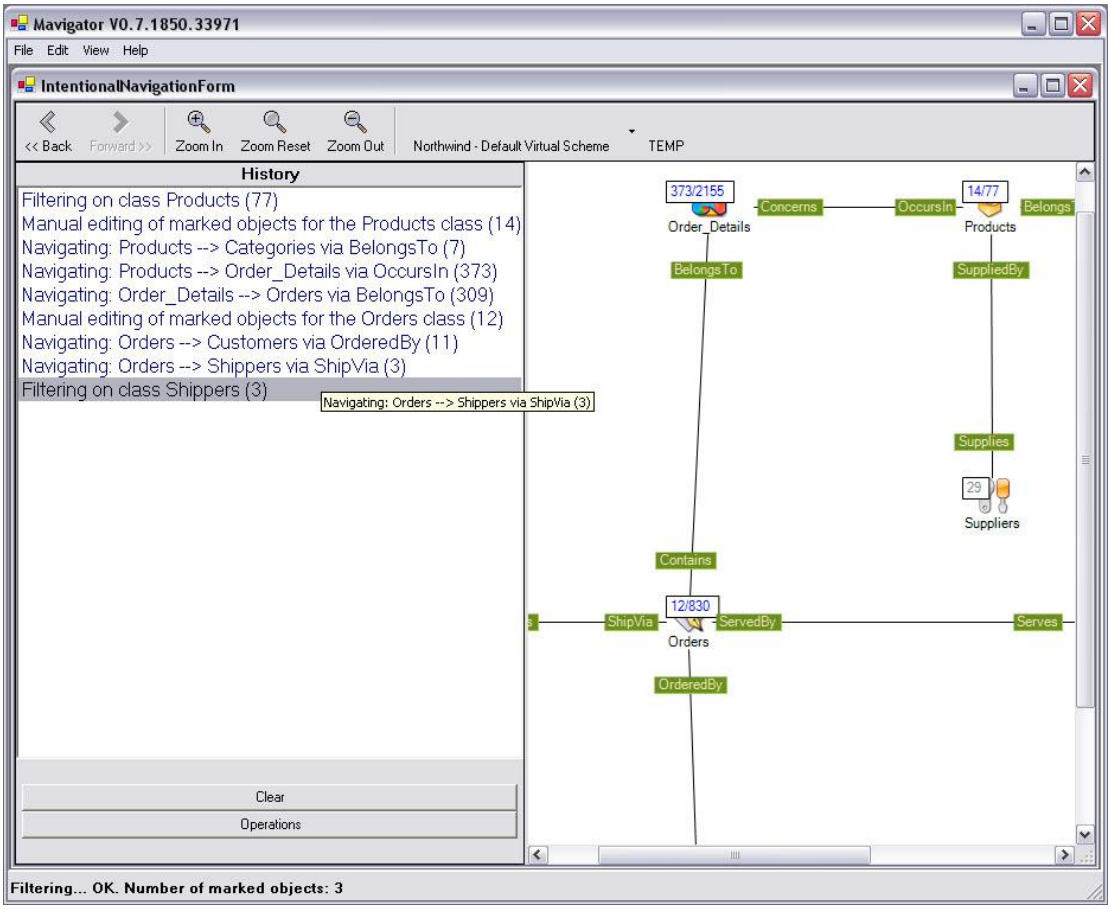

Figure 35. List containing the history of the marking objects.

On the left side of the database schema graph, there is a list (see Figure 35) containing all actions related to the marked objects. The list and the schema are

separated by a widget called splitter. It allows dragging itself, which means changing the space occupied by the list (left part of the Figure 35) and by the schema (right part of the Figure 35). The user can read additional information (or just basic information if it does not fit in the list's space) from the special tool tips shown for each item individually.

The list of marked objects activities (simply called "history") could also be utilized to perform additional actions. Clicking on one of the actions, means undoing all actions from the end up to the clicked one. There is also a possibility to use one of the two buttons: *back*, *forward* (see left part of the toolbar from the Figure 35). Its utilizations are very similar to buttons in Internet browser – clicking *back* moves back to the previous actions, *forward* to the next action (of course only in case of using *back* before).

4.3.4 Showing Objects

Sometimes there may be a need to see a list containing all marked objects. However such an action would be possible only if there will be a way to present "names" of the objects. The problem is how the system will be able to name objects from various classes. Our proposal is based on special method implemented by data source wrapper, which returns dedicated label for each object. Implementations details could be found in Chapter 5.2.

Figure 36. List of marked objects of the Suppliers class.

User of the Mavigator can choose command *Show* from the marked objects' context menu (see menu on Figure 30). As a result special window containing a list of the marked objects is shown (see Figure 36).

Figure 37. Modifying a set of marked objects using objects list.

However this window allows performing more operations than just viewing a list of objects. A user is able to:

- Change the content of the marked objects set in two ways (when 1 or more objects are selected) – see Figure 37:
  - by leaving only selected objects,
  - by unmarking selected objects – removing from the set.

Figure 38. Using list of objects to perform operation with particular object.

- Perform operations (when only 1 object is selected) with a particular object – see Figure 38:
    - Show content of the object as a table containing attributes, their values, object's label and name as well as the icon of the class - Figure 39,
    - Start new extensional navigation,
    - Add to the main (root) basket. Of course an added object (actually, its OID) could be moved to other sub-basket using basket activities (drag and drop).

Figure 39. Content of the object.

4.4 Extensional Navigation

On contrary to the intensional navigation, extensional one works with individual objects (and links) rather then groups of objects (and associations). The user is able to "touch" particular object, its links with other object, see its content, etc. The big problem concerning extensional navigation is legibility of graph. The problem is so serious that the entire user interface is dependent on it.

4.4.1 Discovering Neighbourhood

As it was described in chapter 3.2, the user starts with a specific (selected) object. This object could be chosen from various sources, in particular:

- From a list of marked objects shown in an objects viewer (see Sub-Chapter 4.3.4),
- From a basket, by using:
  - drag & drop technique,
  - dedicated option from the context menu,
- From another extensional navigation session.

The above activity loads direct neighbourhood of the starting object. Direct neighbourhood means all objects are directly linked with it.

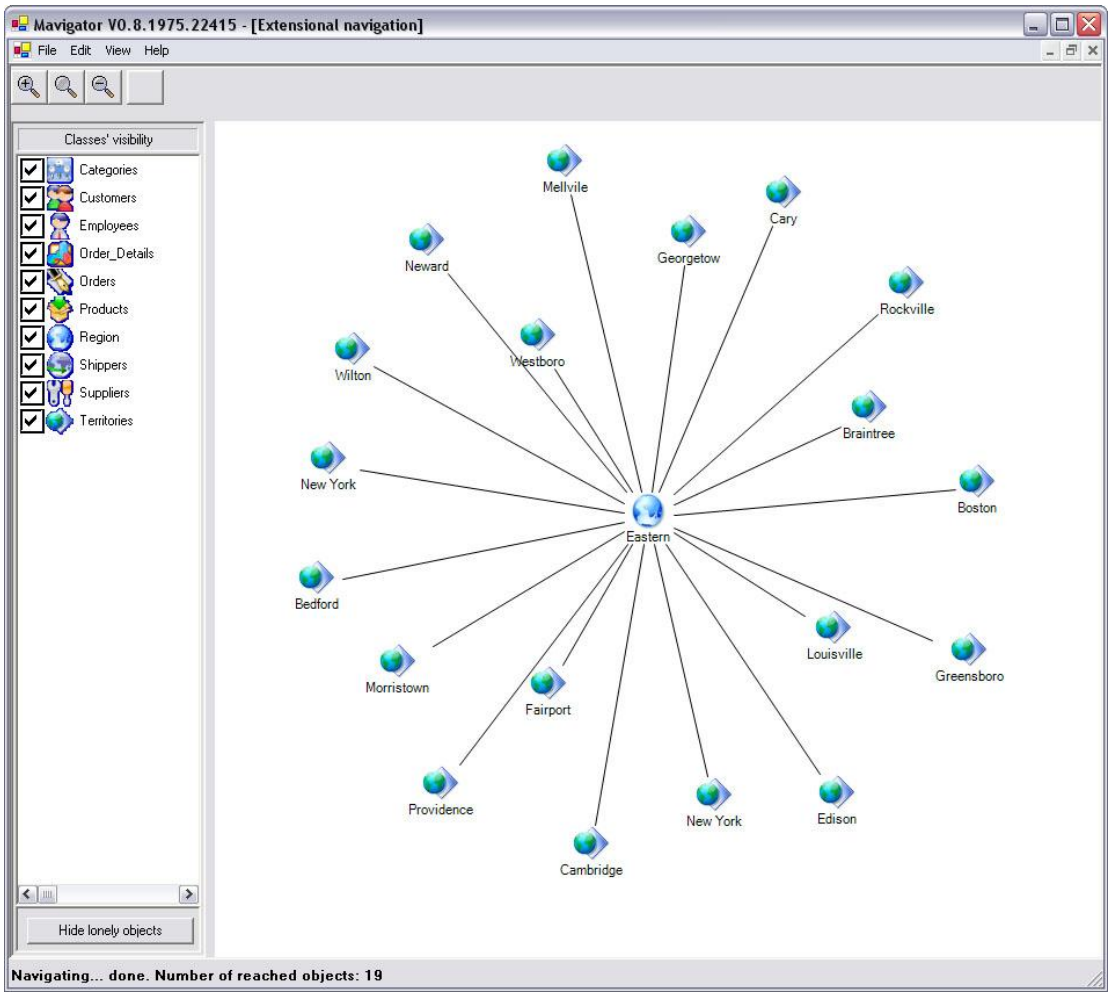

Figure 40. Fist step of the extensional navigation.

Figure 40 shows a first step of the extensional navigation window for *Easter* object (in the centre of the figure) from the *Region* class. As it can be seen, objects are

visualized using their classes icons. It is worth noting that this figure shows direct connection (without an empty, intermediate class *EmployeeTerritories*) between classes *Region* and *Territories*. It has been achieved by using a dedicated virtual schema.

If the user would like to discover next links and objects, all he/she has to do is click the object's icon. The result will be downloading appropriate objects and links from the data source, according to the active virtual schema. For instance, clicking *New York* object downloads employee ("Buchanan"), who manages the territory (see bottom-right of the Figure 41).

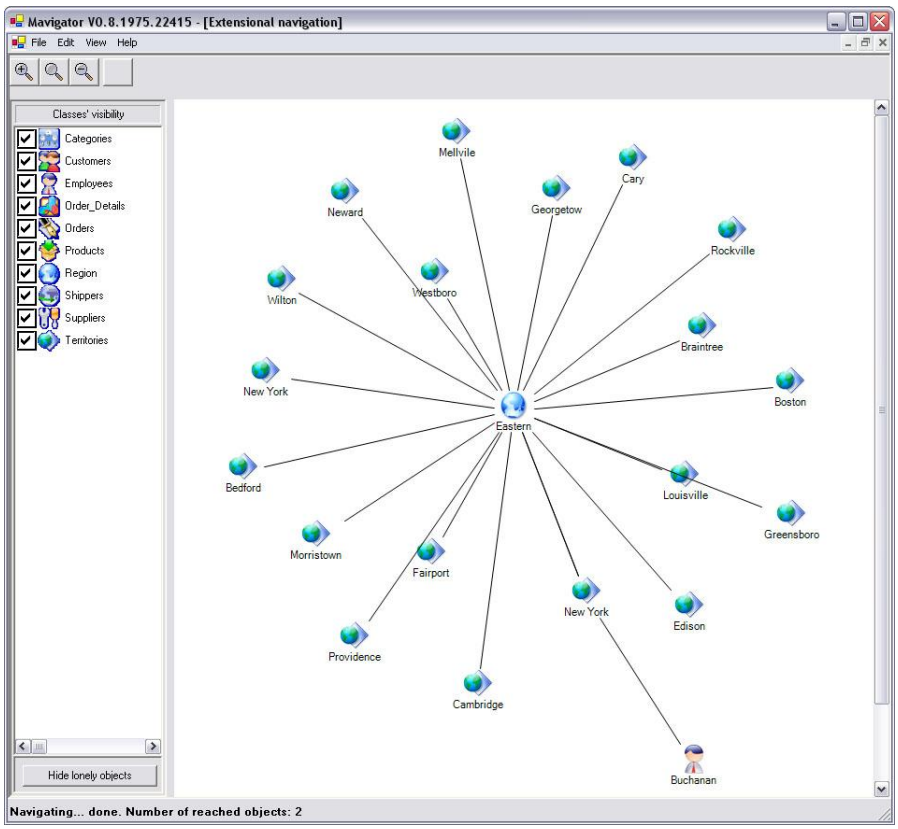

Figure 41. Next step of the extensional navigation.

Of course objects common for several explorations (utilized in various neighbourhoods) are visualized using only one icon. For instance, Figure 42 shows three objects of the *Employee* class ("Fuller", "Peacock", "Buchanan") accessed from various neighbourhoods (i.e. "Buchanan" from "New York" and "Edison"). This approach means using of special implementation techniques – see Chapter 5.4.

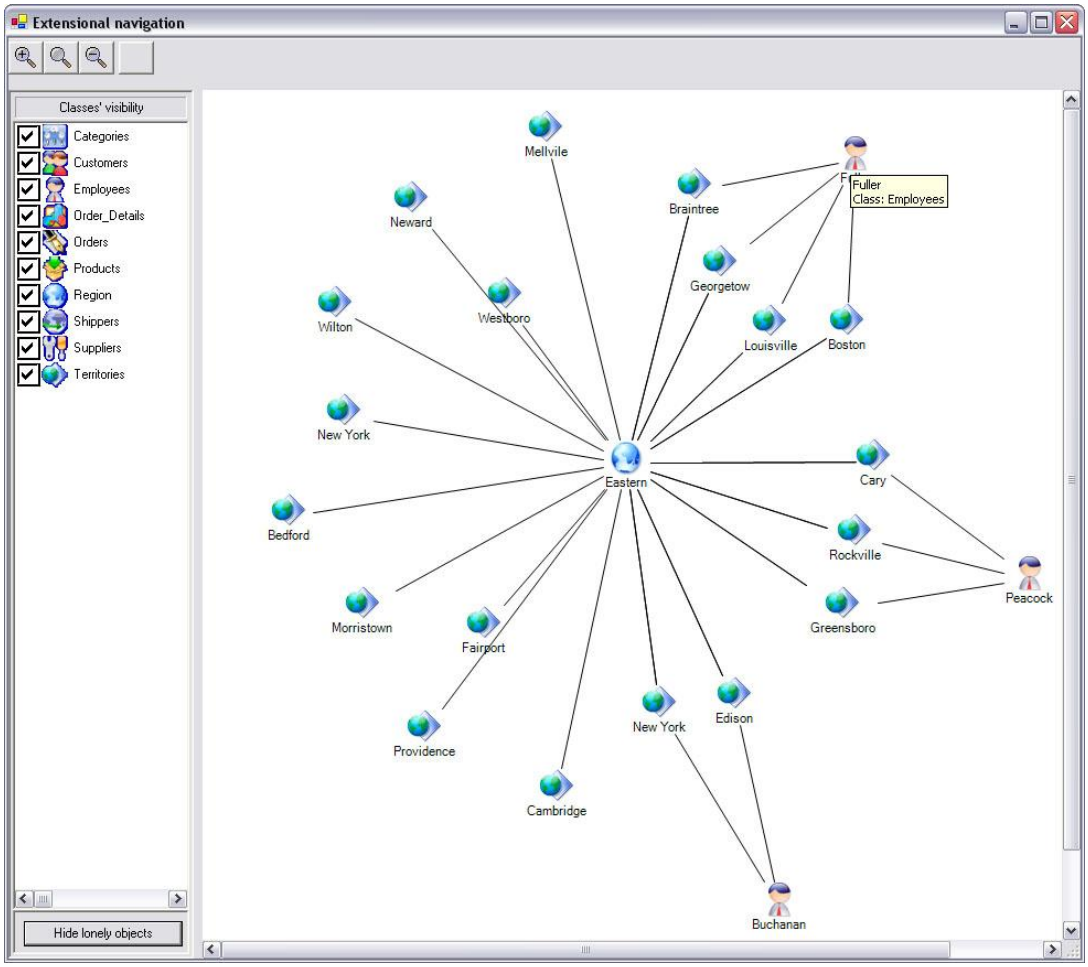

Figure 42. Extensional navigation – common objects accessed from various neighbourhoods.

4.4.2 Layouting the Graph

As we mentioned in the previous chapters, the key issue behind extensional navigation is legibility of the graph. To improve it we have introduced special layouting algorithm, running in the background and automatically arranging objects' icons. As a complement to the above functionality we have added additional functions. The user is able to:

- Manual moving particular objects. Dragging and moving object causes automatic changes the rest of the graph (without dragged object). When the user releases the object, its position will also be smoothly rearranged.
- Pin objects to a particular location (Figure 43). The layouting algorithm does not process objects, which are pinned to the surface. Such objects have special

mark painted on the icon. Using special option from the context menu, the user can un-pin object at any time.

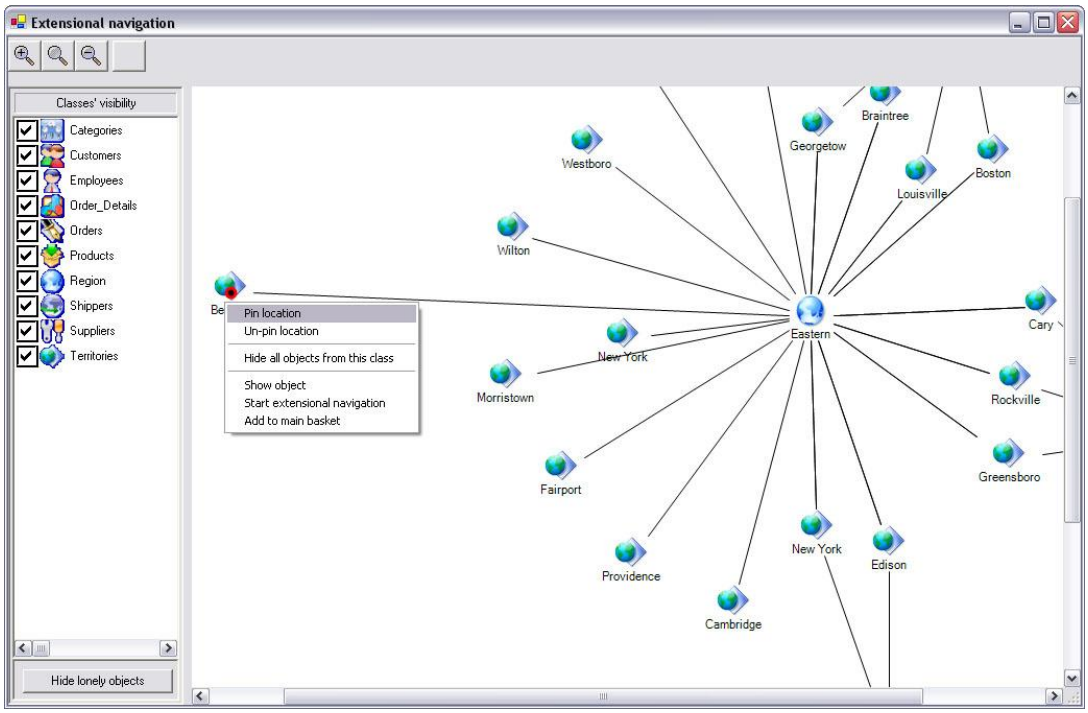

Figure 43. Illustrating of the pinning object.

The important, general contributions to solving legibility problems are using:

- zoom in, zoom out,
- tool tips.

For instance Figure 44 shows zoom in on the part of the extensional navigation session. By reading the information, we can find out that Order number 11000:

- contains following products: “Original Frankfurter grune Sobe”, “Chef Anton’s Cajun Seasoning”, “Guarana Fantastica”,
- has been delivered by “Federal Shipping”,
- and has been ordered by client named “Rattlesnake Canyon Grocery”.

Additional details (i.e. regarding number of items ordered) could be found after showing object’s content (i.e. from *OrderDetails* class).

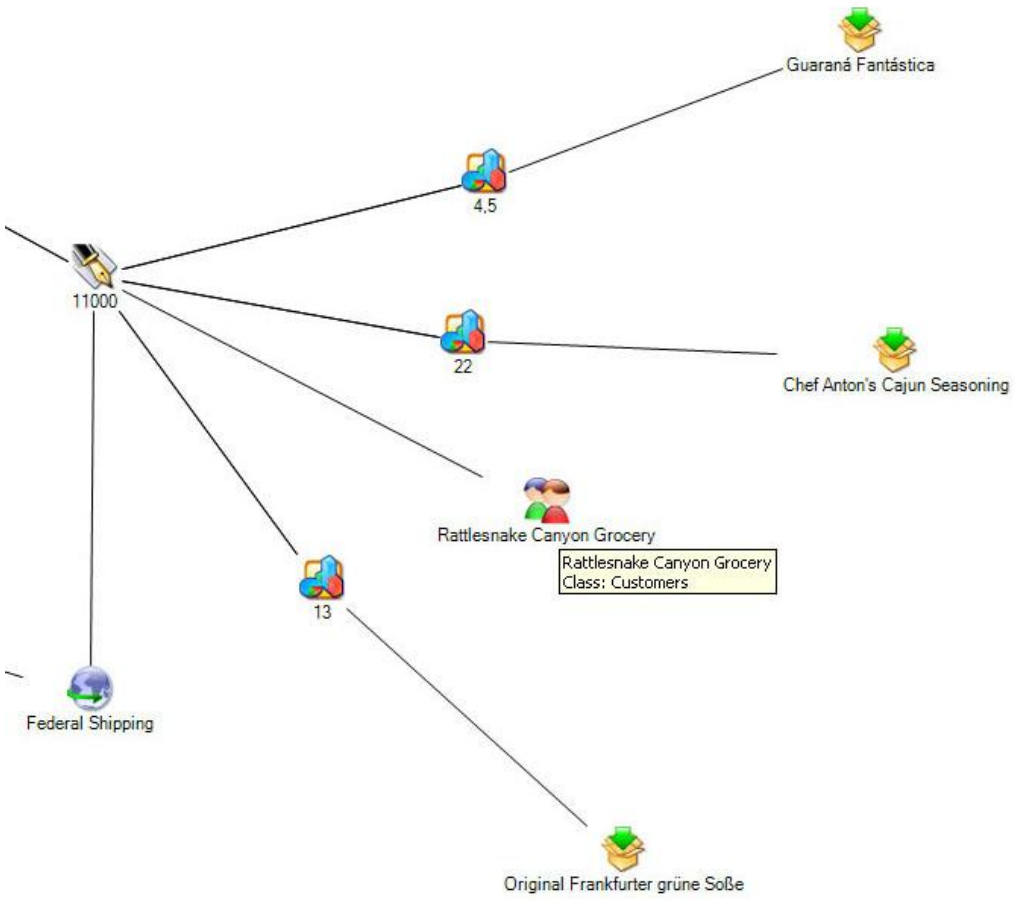

Figure 44. Illustrating of the zoom in during extensional navigation session.

4.4.3 Hiding Objects

As a result of extensional navigations (exploring the neighbourhoods), the graph could have hundreds of objects. Mostly, only some of them are interesting for the user. Hence there should be a way to hide some of them. There is a question how to define objects, which are important to our research. If there will be hundreds of them, we cannot hide them one by one. Our proposal is to hide all objects from particular class. It could be achieved in two ways:

- The first one by using menu on the left side of the window,
- The second one is choosing appropriate option from object's context menu.

The results are identical: all objects from particular class become hidden. To reverse the action, the user has to turn on an appropriate (previously turned off) check box. Such an action may cause creating of isolated objects (not connected with others)

on the diagram (because we have just hide “intermediary” - connecting objects, which occur in hiding links too). It is possible to “clean up” the diagram - there is a special button, which hides such objects.

As a sample, let’s take into account the situation from Figure 42. The user would like to analyze data linked only to the three employees. To achieve that the user should turn off check box near *Region* class (in the list called Classes’ visibility). Figure 45 shows the graph after the operation and hiding isolated objects.

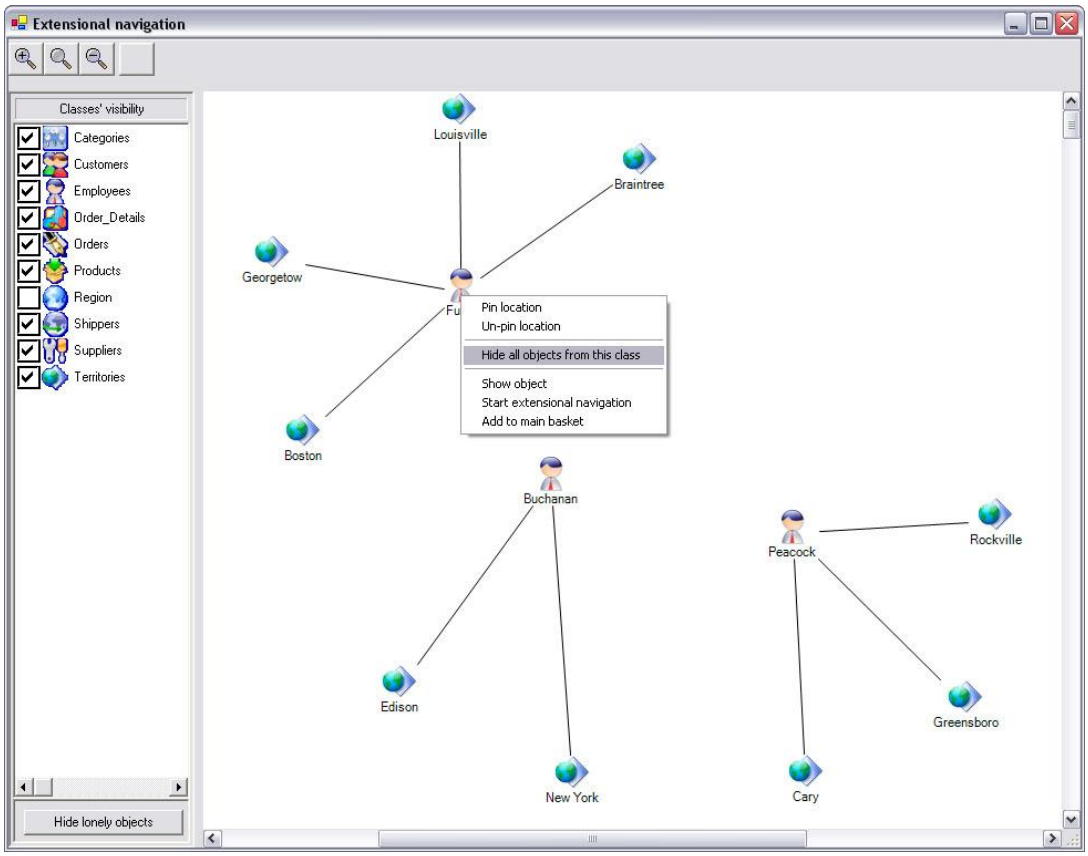

Figure 45. An example of hiding objects from a particular class.

4.4.4 Working with Object

Coherence of the Mavigator’s metaphors assures that all objects, no matter in what context (basket, extensional navigation, etc), are treated exactly the same way. By “having” a particular object, the user has access to a special context menu (the last three positions from the context menu shown on Figure 46) allowing performing some operations (described in details in chapter 4.3.4):

- Showing the content of the object,
- Starting a new extensional navigation,
- Adding to the main (root) basket.

Figure 46. Object context menu.

Besides the above general options, there are some additional operations, accessible only in a specific context, i.e. pining object to the surface in the extensional navigation session.

4.5 Baskets

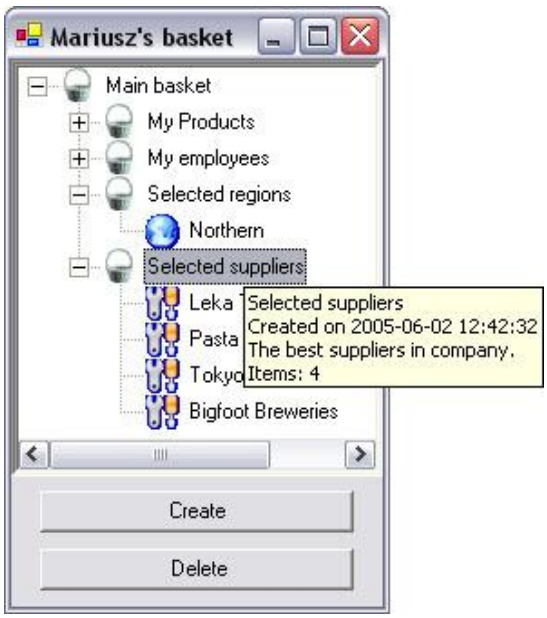

Figure 47. User's basket after logging in.

Baskets act as storing places for objects. Basket Graphical User Interface (GUI) together with Mavigator's metaphors allows performing all operations in an easy and intuitive way.

According to the information given in chapter 3.3 main basket and all its sub-baskets is persistent and assigned to the particular user. Thus, after log in to the system, user's basket is restored and accessible. Figure 47 shows an example of the user's basket just after successful logging into the system. Notice a specialized tool tip showing basket's properties.

Next sub-chapters describe all basket activities from the user interface point of view.

4.5.1 Creating Baskets

Creating a new basket is a very easy process. All user has to do is selecting a parent-basket (otherwise the main basket will be selected) and enter the required data (see Figure 48):

- Name,
- Description.

Figure 48. Creating a new basket.

Basket's date of creation will be added automatically by the system.

4.5.2 Adding Objects to the Basket

There are three ways of adding objects to the baskets:

- During intensional navigation session, user can drag class's icon and drop onto the basket. The action means adding all marked objects, from particular class, to the newly created basket. That is, user is asked (Figure 48) to enter data describing new basket, which improves order inside the basket.
- During extensional navigation, it is possible to drag particular object's icon and drop it onto the basket. As a result, dropped object will be added to the existing basket (that is, without creating new one).
- During any operation on the particular object, user can choose appropriate option (*Add to the main basket*) from the object's context menu – see Figure 46.

4.5.3 Items' Utilization

Objects from the basket could be utilized in two general ways:

- The first one is started from the object's context menu and comprises standard object's operations (Figure 49):
  - Show content of the object,
  - Start extensional navigation with this object,
  - Add this object to the main basket.

Figure 49. Context menu for object in a basket.

- The second way includes object's utilization in the another Mavigator's metaphors:
    - Starting extensional navigation for a particular object. The user can drag object from the basket and put it onto the extensional navigation surface. If there are some other objects (or even dragged object), they will be coherently utilized. The same action could be performed with a whole basket, which means starting extensional navigation for all objects from the basket (and all sub-baskets).

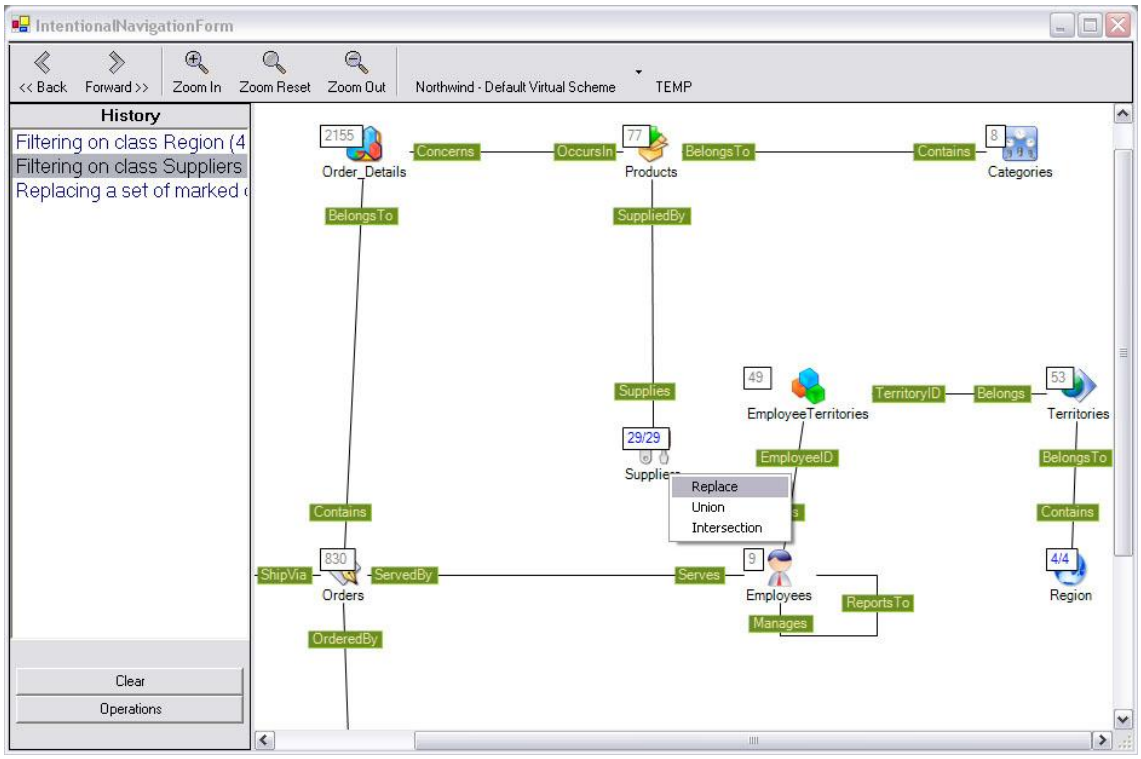

Figure 50. Using basket to mark objects.

    - Marking objects during the intensional navigation session. The user can drag a basket and put it onto class visualization. Processing concerns all objects from particular basket and all its sub-baskets. However only objects belonging to the selected class are taking into account (the rest is ignored). After the drop, there is possibility to choose action to perform on a new set of marked objects (the dropped one) and existing one:
        - Replace existing set with the new one,

- Sum two sets (without repetition),
- Intersect two sets.

For instance, Figure 50 illustrates above operation for the basket *Selected suppliers* shown on Figure 49.

The analogical marking could be performed with a single object rather then with the whole basket.

The second kinds of basket items are sub-baskets, which could be processed as a whole by Active Extensions (see Chapter 4.6). In that case all objects from all sub-baskets are put on the list and then processed. However there might be a problem caused by the multi-class nature of the baskets (objects in the basket could become to many different classes). As a default, an Active Extension is dedicated to process objects from one (particular) class. The situation where processed objects belong to different classes could lead to errors. On the other hand, abandoning multi-class basket nature, would lead to serious restrictions for the user. Thus, an Active Extensions programmer has to take into account that the processed objects could come from different classes.

4.5.4 Basket's Items Properties

Both kinds of items, which are stored in the basket, have properties. Some of them are read-only, others could be changed using dedicated user interface. Below we enumerate all of them, indicating if a property is read-only (R):

- another sub-baskets:
  - creation date (R),
  - name,
  - description,
  - number of items in the basket (R),
- objects:
  - adding date (R),
  - comment, which could be add at any time after placing the object in the basket.

Values of properties are shown using dedicated tool tip. Sample tool tip (for a basket) is shown on Figure 47.

4.5.5 Basket's operations

When the user selects two baskets, there is a possibility to perform three kinds of operations on the baskets:

- Sum,
- Intersection,
- Difference.

The result operation could be stored in another basket (especially created) or one in the participation baskets. Part (a) of the Figure 51 shows summing of two baskets into another one. The user has selected *Best suppliers* and *Tom's suppliers* and then has chosen dedicated option from the context menu. The basket *Merged suppliers* being the result of operation is shown on the part (b) of the Figure 51. It is worth noting that object *Leka Trading*, has occurred in both baskets. However the result contains only one instance.

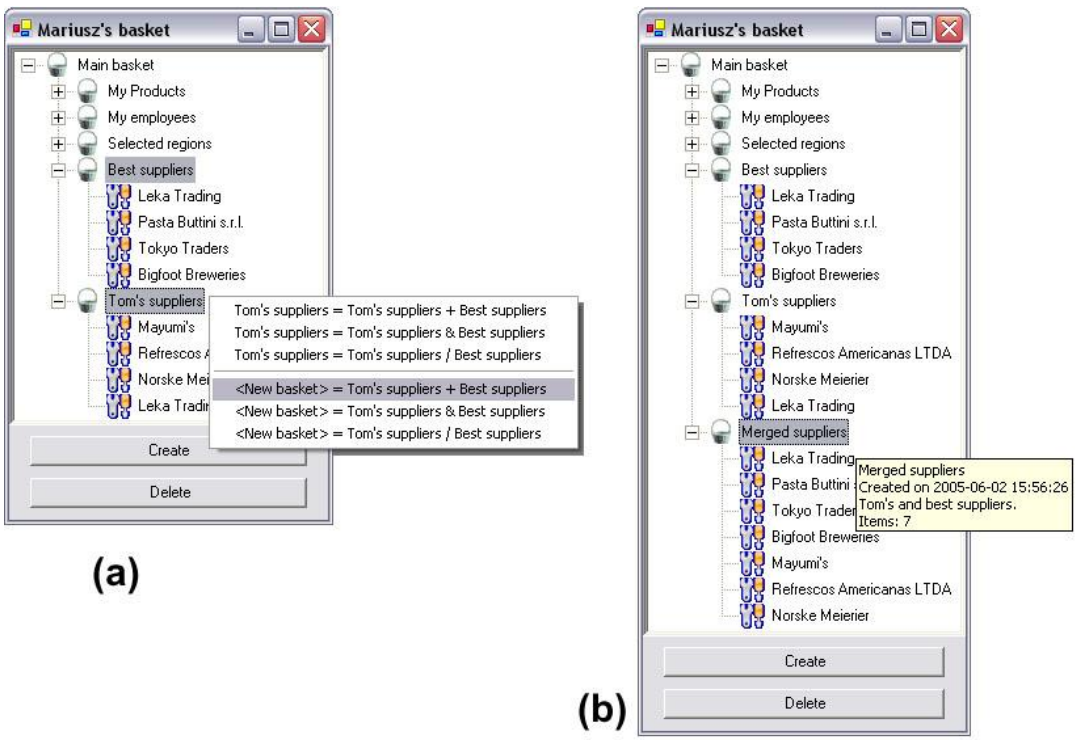

Figure 51. An example operation (sum of two baskets) on baskets.

And there are two more basket's operations from another category. They make easier keeping order in a basket:

- Any object could be moved from one basket to another. It is especially useful when the user uses a context menu option allowing adding object to the main basket. Then after adding, the user can create appropriate sub-baskets and locate particular objects in suitable sub-baskets.
- An object or a sub-basket could be removed from a basket. It is achieved by selecting one and pressing *Del* key or clicking dedicated button in the basket window.

4.6 Active Extensions

In chapter 3.4, we have described Active Extensions (AE), which allow expanding existing Mavigator's functionalities. In a user interface for AE two different groups of functionalities could be distinguished. Next sub-chapters give their detailed description.

4.6.1 Active Extensions Editor

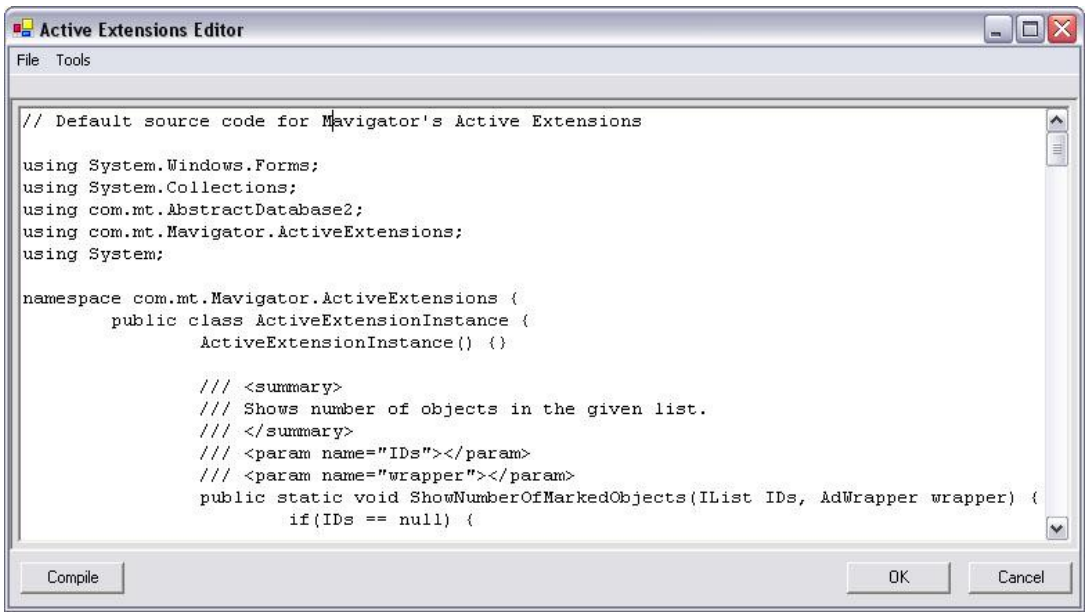

Figure 52. Editor for the Active Extensions.

The Mavigator prototype is equipped with a quite simple Active Extensions editor. The editor is started by an appropriate command from the top window menu and allows:

- writing a source code for the Active Extension,
- loading an existing source code,
- saving a created source code,
- compiling a code.

Figure 52 presents an editor window with a part of the default (shipped with the Mavigator) Active Extensions code. When AE editor is started, it tries to load the source code from the default file located in the special Mavigator's directory. If the file has been found, its content is shown. Otherwise an empty file is generated. The programmer can write any kind of Microsoft C# programs. When he/she wants to try if the code is valid, the *Compile* button must be clicked. As a result, an appropriate message is shown. In case of an error, a simple description (also containing line and column numbers) is presented (Figure 53).

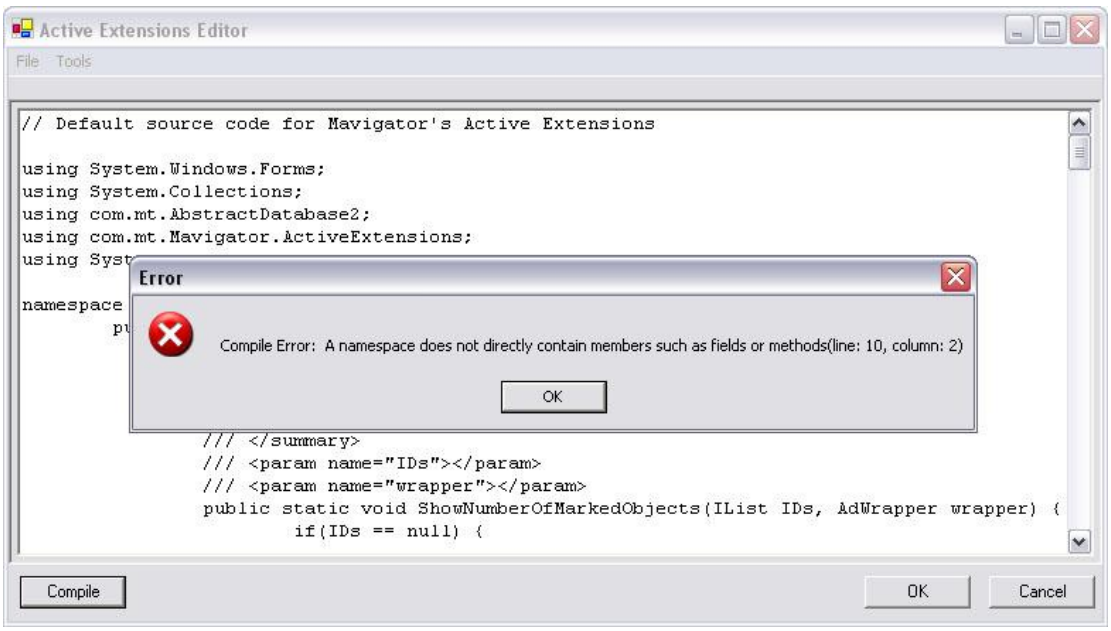

Figure 53. An example message with error description regarding AE source code.

After successfully compilation programmer clicks OK button, which means that all Active Extensions have been registered and are ready to use with one click button.

4.6.2 Using Active Extensions

As we mentioned in chapter 3.4, using of AE is very simple. There are two ways of starting a particular Active Extension. The difference among them is that the first one is running on a basket content and the second one operates on marked objects of a particular class. All the user has to do is choose a class with a set of marked objects (or just a basket) and select one of the available AE from the context menu (Figure 54). The rest of actions depend on a particular AE.

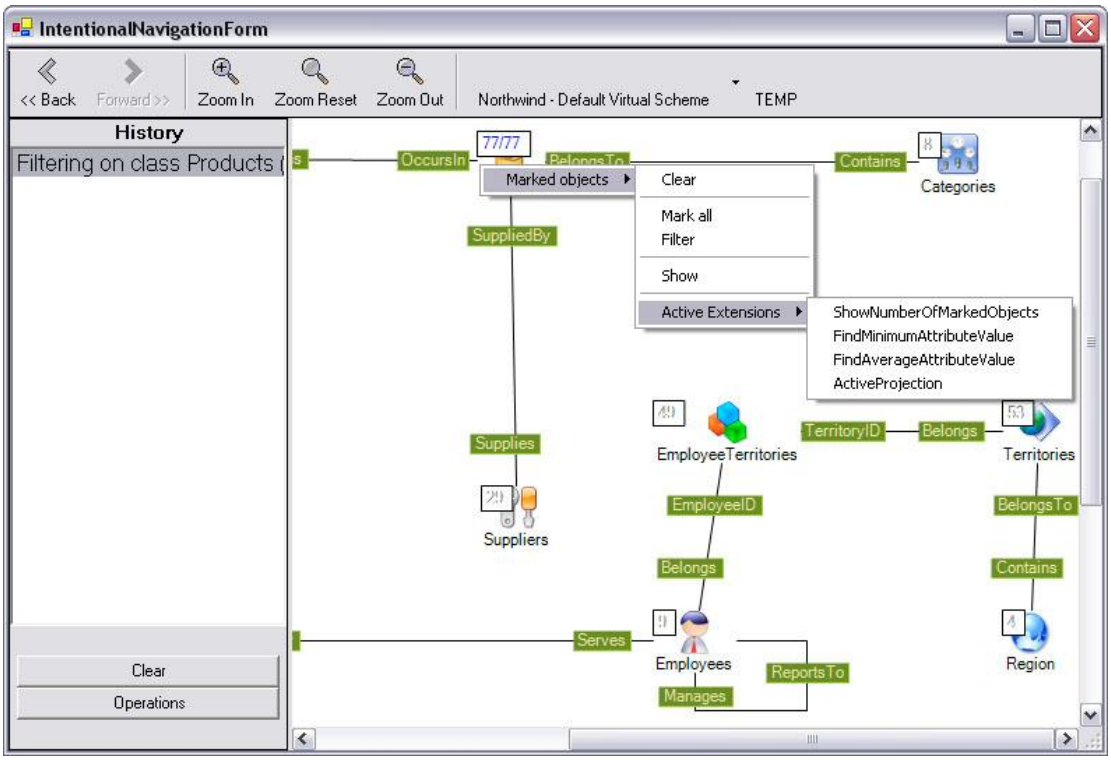

Figure 54. Starting an active extension.

The implemented sample set of Active Extensions contains five entries. Three of them are quite simple and thus will be described in one sub-chapter. The fourth and fifth are more advanced and will be presented in the dedicated chapters.

4.6.2.1 Simple Active Extensions

The simplest AE, developed in educational purposes only (for programmers, who want to create own AEs), only shows the number of marked objects.

Next two AEs are more useful:

- The first finds the object with the minimum value of the chosen attribute,
- The second one calculates the average value of the selected attribute.

All calculations are performed on marked objects or on basket content. In both cases, the user only selects an attribute (part (a) of the Figure 55), which takes a part in calculations. The result for the first calculation (the minimal value of the attribute) and a label of the appropriate object shows part b of Figure 55.

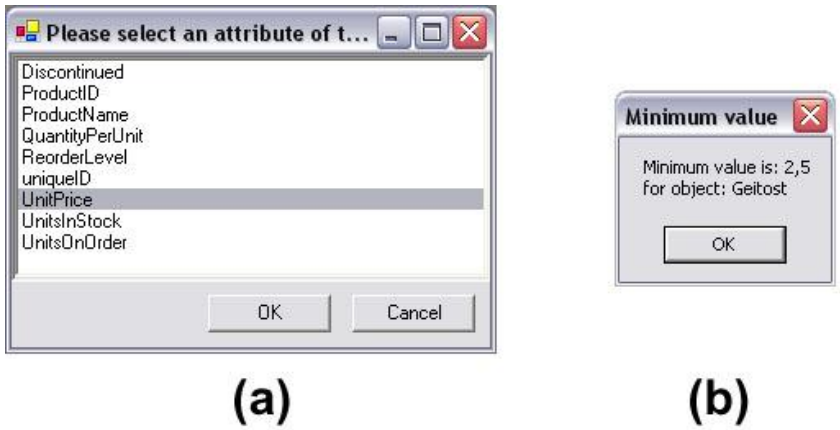

Figure 55. Selecting an attribute for the calculation (a) and the result (b) containing the value and object's label.

4.6.2.2 Active Projections

Another Active Extension developed with the Mavigator prototype is Active Projections. They allow projecting (showing) objects on the surface, where coordinates of the particular object depend on values of selected attributes. The approach makes easy performing some analysis. In general, such projections could visualize dependencies between many attributes. Their numbers start from two or three attributes (using just 2 or 3 axis) up to 5, 6 or even more (using additional, special notation, i.e. colors, shapes, sizes, etc). Current implementation of the Active Projections is able to visualize dependencies between values of two attributes.

After starting the extension (exactly the same way like in the simple ones), the user has to choose two attributes, one by one. The manner of selecting the attributes is exactly the same like in the one described in chapter 4.6.2.1 and shown on part b of the Figure 55. As a result, after a short while (depending on the number of objects), the user can see the projection. Sample projection for *Products* class and attributes *Unit Price* and *Units in Stock* is shown on Figure 56. Notice the special tool tip showing the label and values of the selected attributes for particular object.

Figure 56. Active Projection for some objects of the Products class.

Because of coherent cooperation of all Mavigator's metaphors, objects projected on the surface, act as objects utilized in another sessions. The user can utilize the context menu (shown on Figure 57), where there are two specialized options:

- Remove object from projection, which means just removing from the visualization,

- Remove from the set of marked objects. The option could be used as an additional way of modifying the set of marked objects.

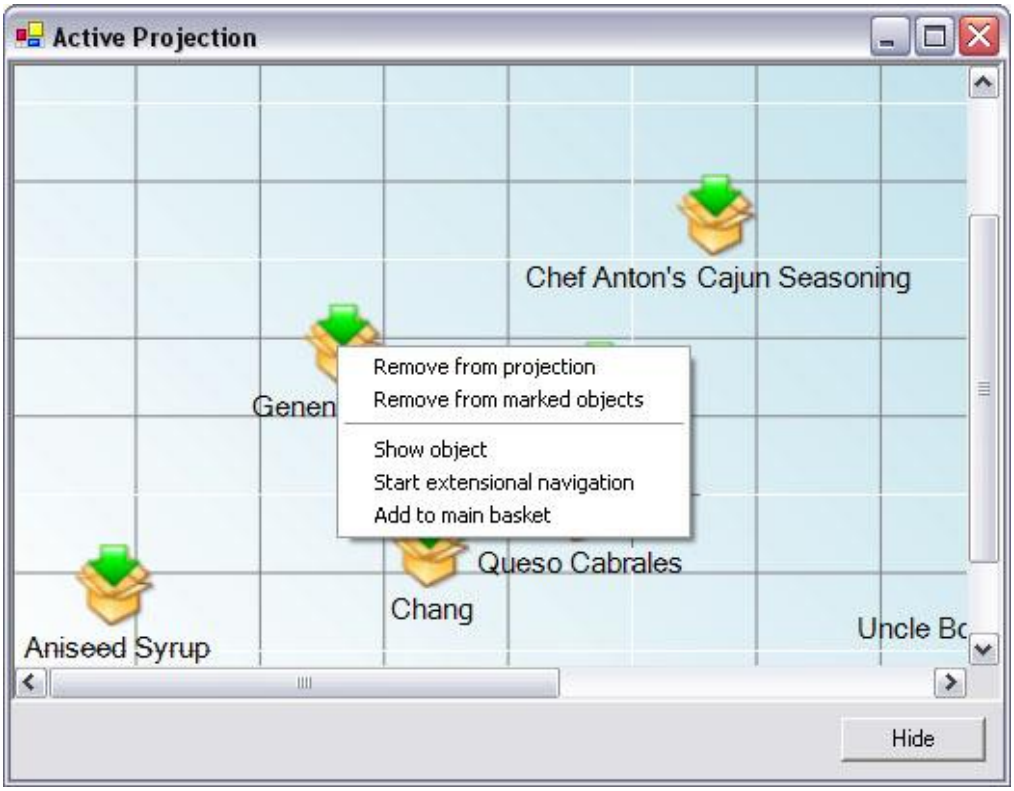

Figure 57. Object's context menu in Active Projections

Besides above commands, there are common objects operations described in details in Chapter 4.4.4.

When a projection is created, the user can add new objects in an easy and intuitive way. All he/she has to do is drag an object (from a basket or extensional navigation surface) and drop onto the projection surface. The same action is also possible for the whole basket (which means adding a group of all objects from the basket and all sub-baskets). If a dropped object (or objects) belongs to the projected object class, it will be placed according to the attributes' values. Otherwise, the dropping will be ignored.

It is also possible to perform reverse actions: drag an object and drop onto:

- An extensional navigation surface, which starts extensional navigation for this object and objects already present on the surface,
- A basket, which simply adds the object to the selected basket.

4.6.2.3 Objects' Exporter

This extension allows exporting objects to an XML file. From the user point of view, the whole process is straightforward and contains only a few elements, namely:

- first the user has to choose values of which attributes should be exported,
- then a filename should be given,
- after successful exporting an appropriate message is shown to the user.

Created XML file has very simple format (Example 15), which allows importing them by almost all nowadays applications.

```
<!--Objects exported by Mavigator V0.8.1980.33966-->
<!--Generated on: 2005-06-03 19:52:49-->
<Objects>
  <Object>
    <ProductName>Chai</ProductName>
    <UnitsInStock>39</UnitsInStock>
    <UnitPrice>18</UnitPrice>
  </Object>
  <Object>
    <ProductName>Chang</ProductName>
    <UnitsInStock>17</UnitsInStock>
    <UnitPrice>19</UnitPrice>
  </Object>
  <Object>
    <ProductName>Aniseed Syrup</ProductName>
    <UnitsInStock>13</UnitsInStock>
    <UnitPrice>10</UnitPrice>
  </Object>
</Objects>
```

Example 15. File generated by ExportObjects active extension.

It is worth noting that an objects exporter could be used in Mavigator's data retrieval activities, which are answers to textual queries asking for values of some attributes (rather then entire objects), i.e. name and address of the employees performing specified conditions (see Example 2 on page 7, Example 3 on page 9).

4.7 Supporting Techniques

In this sub-chapter we would like to enumerate some items of the graphical user interface, which assures comfortable work with the Mavigator prototype. They are not necessary from the utilized metaphors point of view, but greatly enhance user's awareness. They contain:

- Progress indicators (see Figure 58). All actions, which could take quite long, including:
  - connecting to the data source,
  - filtering data,
  - navigating,
  - showing object's content

  are illustrated with the animation showing rotating hourglass and appropriate message. This approach assures that the user will see that the operation is in progress.

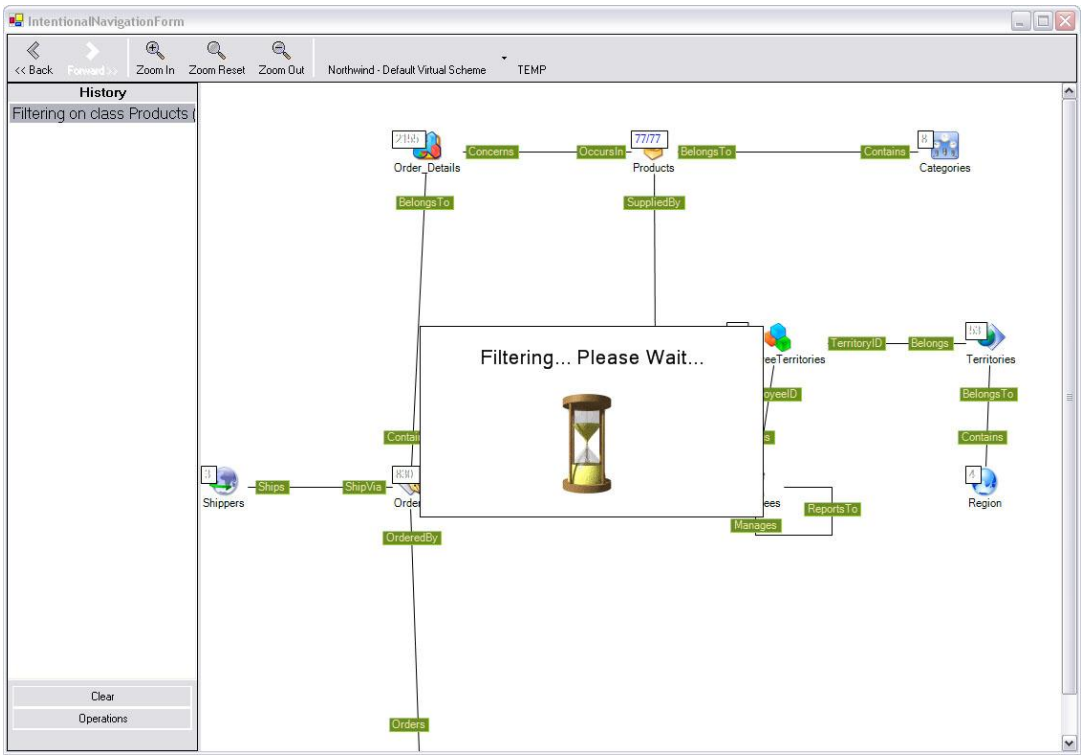

Figure 58. Widget illustrating that operation is in progress.

- All surfaces utilized in Mavigator, including:
  - intensional navigation,
  - extensional navigation,
  - active projections

  allows zooming in and out. The operations could be performed using dedicated buttons or simply a mouse wheel.
- Wherever it is possible, specialized tool tips are used. Depending on context, they show different kinds of information, see examples on Figure 44, Figure 47 and Figure 56.
- The status bar showing results of the user actions.

It is worth to notice that all of them have needed special programmatic effort, especially in case of a progress widget (because they require introducing threads to the system's design, which is always complicated).

4.8 Case Studies

This chapter contains samples of the Mavigator's utilization. We tried to create examples, which would be:

- based on the Northwind database (and a virtual schema hiding intermediary class *EmployeeTerritories*), because this is the database commonly available,
- similar to those ones shown in the Chapter 2.1. Such an approach allows comparing the Mavigator usability potential to the other systems.

Next sub-chapters give detailed description of Mavigator's user's actions according to case studies. All following solutions assume that a user is logged to the system. It is worth noting that thanks to flexible metaphors, most of the cases could be solved in many ways. A presented way is just one of the few possible.

4.8.1 Case 1

Find all employees with first name "Robert".

Actions:

- Start intensional navigation (from top window menu),
- Chose *Filter* from the marked objects' context menu for *Employees* class,
- Define appropriate filtering criterion containing one predicate (shown on Figure 59)
- Start filtering by pressing *Filter* button. As a result, set of marked objects for the Employee class will contain all objects describing employees with first name "Robert". In case of Northwind database it will be only one object.

Figure 59. Filtering criterion for case study 1.

4.8.2 Case 2

Find all employees with first name "Robert" or "Nancy".

Actions:

- All actions are exactly the same like in the case 1, except the filtering criterion contains 2 predicates connected with OR (Figure 60).

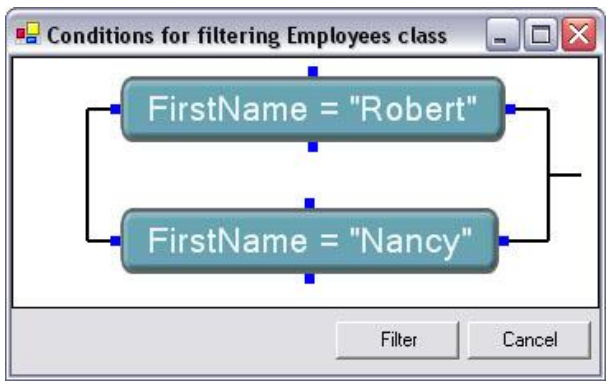

Figure 60. Filtering criterion for case study 2.

4.8.3 Case 3

Find the name of the employee who manages territory "Cambridge".

This case study is similar to the OQL one defined in Example 1 on page 7.

This case requires cooperation between two classes (*Employee* and *Territory*). Actions:

- Start intensional navigation,
- Filter objects from *Territory* class (similarly like in case 1) to find territory "Cambridge",
- Navigate from marked objects of *Territory* class through *ManagedBy* role (click on role's name and select *Replace* previous set of marked objects for the *Employee* class). As the result, a set of MO for the *Employee* class, will have 1 object named "Fuller".
- As an alternative to the above action, the user can start extensional navigation for the "Cambridge" object (from object's viewer using context menu) and just see all connected objects, including employee, who manages "Cambridge" (Figure 61).

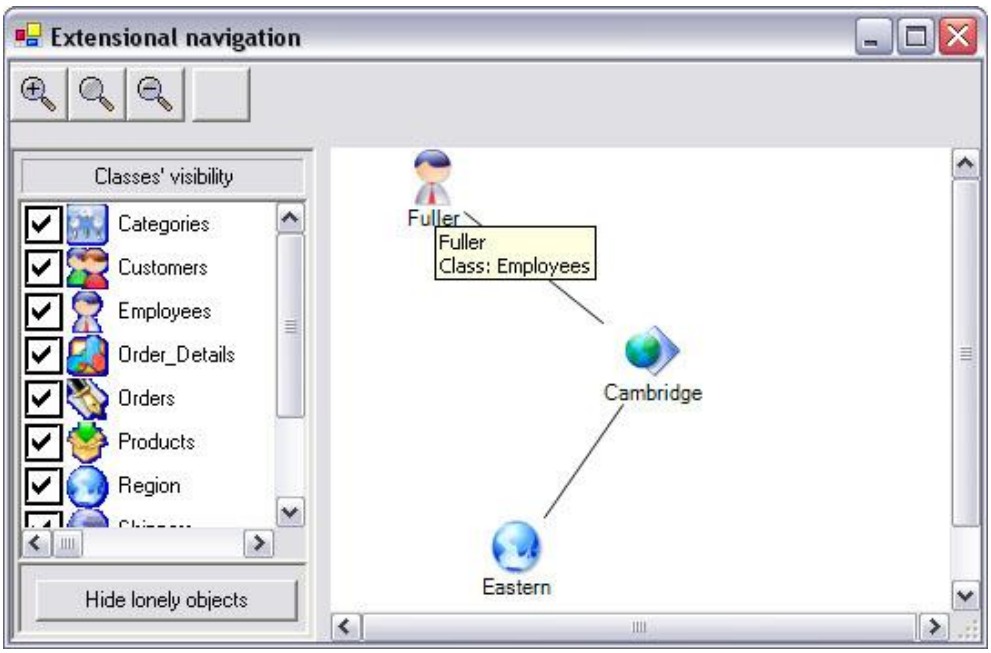

Figure 61. Extensional navigation session for case 3.

4.8.4 Case 4

Find names and prices for all products from supplier "Leka Trading" where prices are more than $18.00.

This case study is similar to the one defined in Example 2 on page 7.

Actions:

- Start intensional navigation,
- Filter objects from *Supplier* class (similarly like in case 1) to find supplier "Leka Trading",
- Navigate from marked objects of *Supplier* class through the *Supplies* role (click on role's name and select *Replace* previous set of marked objects for *Products* class). As the result, set of MO for the *Products* class, will have 3 objects (all from the supplier).
- Filter existing set of marked object to find products with price higher than $18.00. As a result two objects will be marked.

```
<!--Objects exported by Mavigator V0.8.1980.33966-->
<!--Generated on: 2005-06-04 19:42:05-->
<Objects>
  <Object>
    <ProductName>Ipoh Coffee</ProductName>
    <UnitPrice>46</UnitPrice>
  </Object>
  <Object>
    <ProductName>Gula Malacca</ProductName>
    <UnitPrice>19,45</UnitPrice>
  </Object>
</Objects>
```

Figure 62. Content of the file being solution to the case 4.

- If the user really needs only names and prices (instead of the entire objects), dedicated active extension could be utilized: Export Objects. Do the following:

- Start (from the marked objects' context menu) active extension called Export Objects,
- Choose desired attributes, XML file name and click Export button.
- As a result, the file will be created containing values of the attributes (see Figure 62).

4.8.5 Case 5

Find employees who served orders sent to the Mexico and manage territories located in the "Northern" region.

This case study is similar (maybe a little bit advanced because requires cooperation of three classes) to the one defined in Example 6 on page 11.

Actions:

- Start intensional navigation,
- Filter all objects from the *Orders* class to find all sent to "Mexico" (ShipCountry = "Mexico"; 28 orders),
- Navigate from *Orders* to the *Employees* via *ServedBy* (from the menu select *Replace*). As a result all employees, who sent orders to the Mexico are in the set of marked objects (of *Employee* class; 7 employees).
- Filter regions to find "Northern" (1 region),
- Navigate from *Region* to *Territories* via *Contains* (11 territories),
- Navigate from *Territories* to *Employees* via *ManagedBy* and select (in the navigation menu) *Intersection* (1 employee - "Callahan")
- Because of selecting *Intersection*, new set of MO (containing all employees, who manage regions from the northern region) will be intersected with the previous set of MO (containing all employees who sent orders to the Mexico) and will contain four employees. Figure 63 shows part of the database graph during described session. Notice the history (left side of Figure 63) containing all operations performed on marked objects.

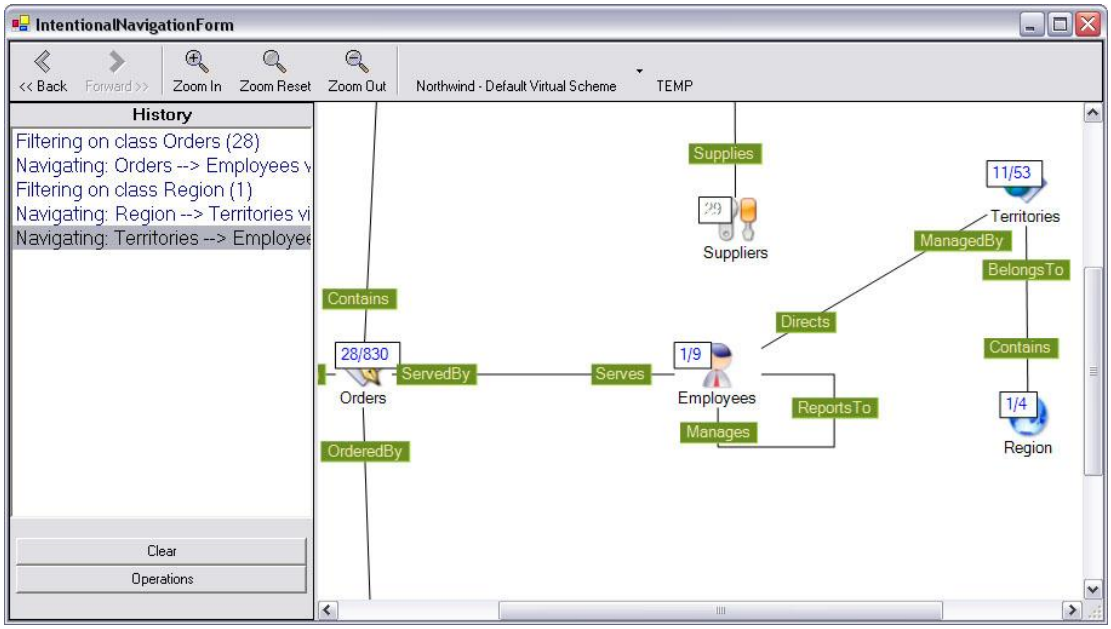

Figure 63. Intensional navigation session during processing case 5.

4.8.6 Case 6

Find average price of the products sent to the Mexico.

Actions:

- Start intensional navigation,
- Find (filter) all Orders sent to Mexico (ShipCountry = “Mexico”; 28 orders),
- Navigate from *Orders* to the *Order_Details* via *Contains* (from the menu select *Replace*). As a result all details (objects) concerning orders sent to Mexico are in the set of marked objects (of *Order_Details* class; 72 objects).
- Navigate from *Order_Details* to *Products* via *Concerns* (from the menu select *Replace*). As a result all products (objects of the *Products* class) sent to the Mexico are marked (45 objects).
- For the *Products* class start Active Extension called *Find Average Attribute Value* and select *Unit price* attribute. The average value is $30.71.

4.8.7 Case 7

Find the cheapest product among those ones, which are in stock.

Actions:

- Start intensional navigation,
- Find all products, which are in stock –filter with criterion containing only one predicate shown on Figure 64.

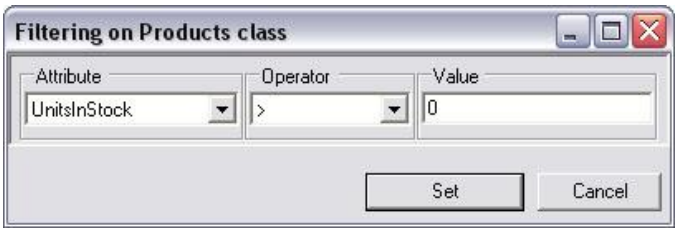

Figure 64. Filtering predicate for case 7.

- For the *Products* class start Active Extension called *Find Minimum Attribute Value* and select *Unit price* attribute. The product is called "Geitost" and the price is $2.50.

4.8.8 Case 8

Analyze the "Seafood" products, to find out which of them are not so much on the stock.

This case study is a little bit different to the previous ones because, the solution is not so deterministic. This is caused by the terms "analyze" and "not so much", which can't be communicated to the system. Of course, there is a possibility to formulate query based on some threshold value (i.e. "not so much" means less than 20). However the better solution could utilize objects' projections.

Actions:

- Start intensional navigation,
- Find "Seafood" object from the Category class (using filtering),
- Navigate from *Categories* to *Products* via *Contains* (from the menu select *Replace*). As a result all Seafood products have been marked (12 objects).
- For the *Products* class start Active Extension called *Active Projections* and select *Unit price* and *Units in Stock* attributes. Part of the projection (zoomed

in) is shown on Figure 65. It can be seen (using tool tips) that stock for "Rogede sild" is only 5 and for "Konbu" is 24.

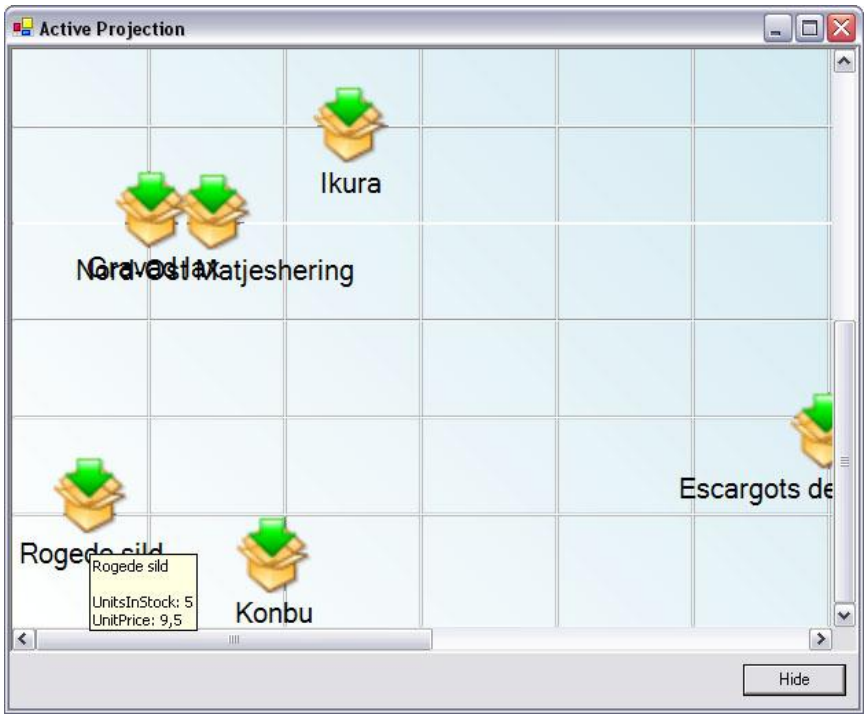

Figure 65. Active projection for case study 8.

4.8.9 Remaining Cases

Figure 66 presents a table containing references to the examples from the chapter 2.1, which could be also solved using techniques described in previous cases.

Example	Page	Comments
Example 3. OQL query selecting name and age of the people.	9	Like in case 1.
Example 4. OQL statement finding people whose age is less than 20.	10	Like in case 1.
Example 5. OQL query finding all people working in companies located in England.	10	Like in case 3.

Figure 66. Table with Mavigator's solutions to the OQL examples from the Chapter 2.1.

5 Software Architecture and Implementation

This chapter gives detailed description of the Mavigator's architecture and implementation. The discussed issues are illustrated with UML classes and sequences diagrams. Some solutions also present parts of the Mavigator's source code (n Microsoft C#). The whole chapter is divided into sub-chapters, which are organized as follows:

- at the beginning there is a general description of the Mavigator's architecture (chapter 5.1),
- next sub-chapters discuss solutions dedicated to the particular topics, namely:
  - Wrapper to the Data Source – chapter 5.2,
  - Virtual Schemas – chapter 5.2.5,
  - Intensional Navigation – chapter 5.3,
  - Extensional Navigation – chapter 5.4,
  - Baskets – chapter 5.5,
  - Active Extensions – chapter 5.6.

And at the end of this introduction, one more thing must be stressed: when designer or programmer develops a new computer system, aside of many other dilemmas, one is especially significant: do we want a good performance or clear internal design? In commercial systems, the first choice would be selected. However because of the nature of this thesis (research), in most cases, we tried following the second approach: clear and easy to understand the design and implementation.

5.1 Mavigator's Architecture

As we mentioned in Chapter 4.1, there are two Mavigator's prototypes. We also stressed that the first one is much simpler (without Active Extensions and Virtual Schemas) than the second one and thus will not be discussed in the dissertation. Therefore we will focus only on the new one developed particularly for the purposes of the present thesis.

The Mavigator prototype is implemented as a Windows Form Application in C# language. Its architecture (Figure 67) consists of the following elements:

- Core GUI – contains implementation of the core user interface elements like:
  - intensional and extensional navigation windows,
  - basket window,
  - Active Extensions editor, etc.
- Business logic – includes implementation of the Mavigator metaphors and some additional routines,
- Active Extensions GUI – GUI elements being a part of the Active Extensions (see Chapter 3.4) like Active Projections window,
- Active Extensions – elements compiled from the source code written by Active Extensions programmer. Arrow, which come from business logic block symbolizes query results processed by AE,
- Database wrapper – ensures communication, via defined `AbstractDatabase2` interface, with any data source. Two things are worth noting:
  - all internal data processing (including Active Extensions) works with abstract data types, which ensures that entire application will be working in the same manner aside of the current data source,
  - moreover, the entire application works with Virtual Schemes (see 3.5). All internal actions related to the Virtual Schemas are realized inside the wrapper.
- Data source. Currently we are working with the ODRA prototype database management system (which follows stack-based approach to query languages [Sub95], [Sub04]), however after implementing dedicated wrapper it is possible to work with any kind of data source: object/relational databases, ODBC, JDBC, etc.

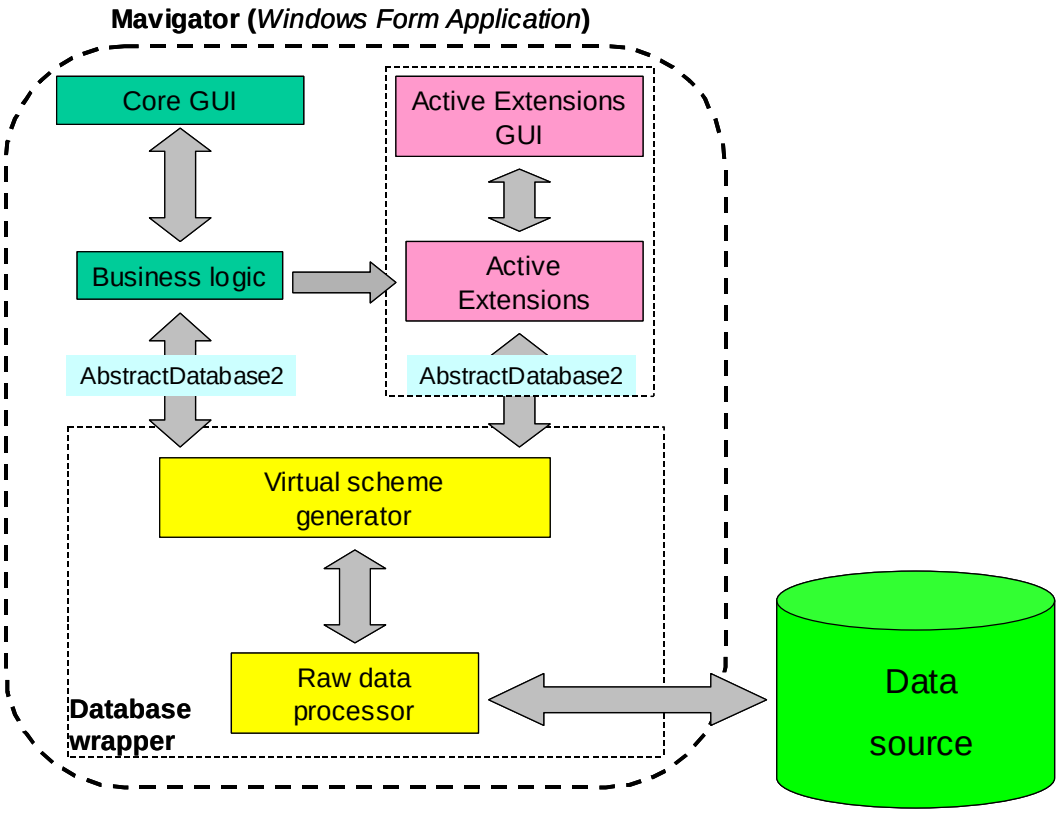

Figure 67. Architecture of the Mavigator prototype

5.2 Wrapper to the Data Source

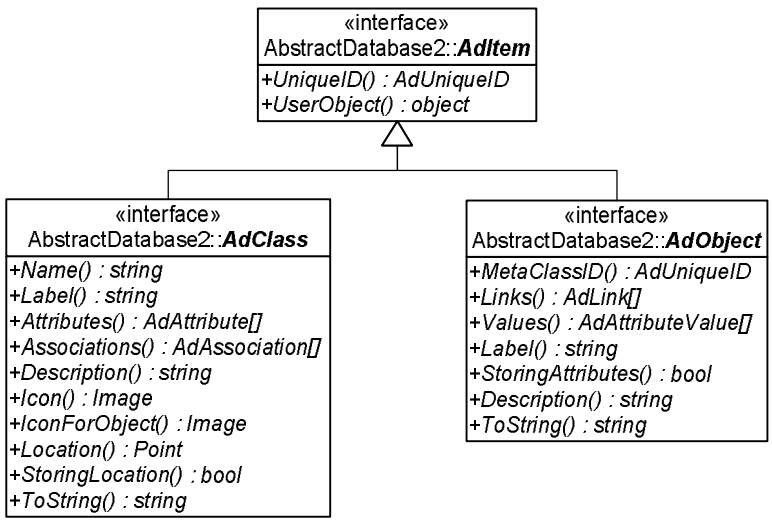

Figure 68. Class diagram for interfaces: `AdClass`, `AdObject`.

The best way to create a system, which will be independent from the connected data source, is develop a wrapper between a data and the system. The wrapper should receive data in some particular format (depending on the data source type) and send them to the system in some precisely defined, generic one. That approach guarantees, that entire system ("behind" the wrapper) will be able to work exactly the same way in any kind of data source (after implementing particular wrapper). All data exchanged between database and the Mavigator are defined in the namespace of interfaces called `AbstractDatabase2`. A programmer, who would like to create a new wrapper, has to implement all these interfaces. From the Mavigator's core functionality point of view, all operations are performed on members of the interfaces' group.

There are more then 10 interfaces defined in the `AbstractDatabase2` namespace. They describe every aspect of data management including:

- classes (`AdClass`) and objects (`AdObject`) - Figure 68,
- associations (`AdAssociation`) and links (`AdLink`) - Figure 69,
- attributes (`AdAttribute`) and their values (`AdAttributeValue`) - Figure 70,

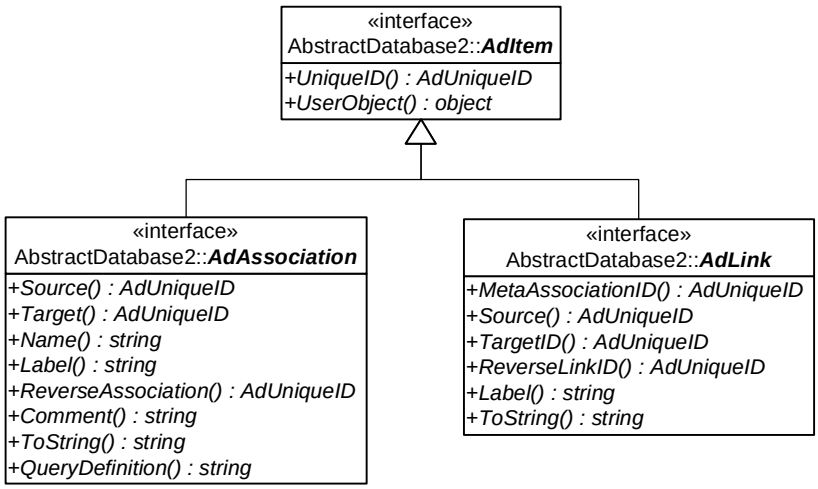

Figure 69. Class diagram for interfaces: `AdAssociation`, `AdLink`.

All above interfaces inherit from the base interface `AdItem` defining only two properties:

- `UniqueID`, which returns instance of the `AdUniqueID` for particular item,

- `UserObject`, which returns any kind of object associated with the item.

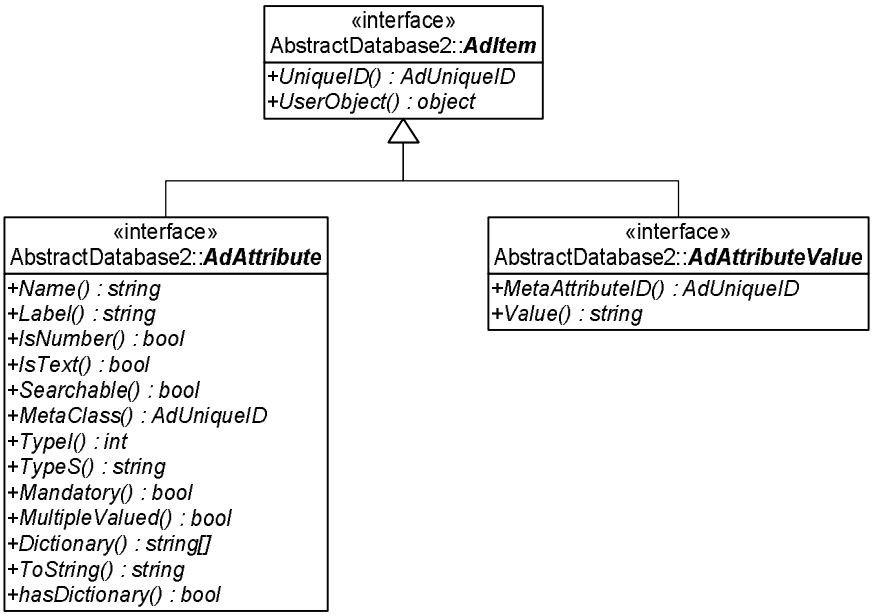

Figure 70. Class diagram for interfaces: `AdAttribute`, `AdAttributeValue`.

The next sub-chapters (up to the 5.2.4) present information without going into implementation details. The details are given in sub-chapters of chapter 5.2.5, treating of Virtual Schemas. The reason is that almost all wrapper's aspects conform to the topic. Thus, they are discussed after presenting, and in the context of, Virtual Schemas.

5.2.1 Unique Objects' Identifiers

Every piece of information (object, class, link, etc), which is processed in the Mavigator, must be uniquely identified. It is achieved by instances of the class implementing `AdUniqueID` interface. In case of the ODRA wrapper, there is a class `AdOdraUniqueID`, which implements the interface. Inside the class there is a private attribute of type `String`, which has unique values among all data coming from the wrapper. The way of calculating its value will differ depending on type of data source and particular information type (object, attribute's value, etc). For instance, in our wrapper:

- objects have numbers, which come from the ODRA database,
- associations have special string being concatenation of class's name, special constant and association's name.

There is one more thing worth noting: the mentioned class has overloaded two methods responsible for comparing objects' instances (`bool Equals(object id)`, `int GetHashCode()`). Thanks to the solution, comparing instances of identifiers is based on its values and not on C# memory references. Thus, different C# instances of the `AdUniqueID`, identifying the same item, are equal.

5.2.2 Wrapper Object

The main object, which deals with communication to/from data source, is an instance of the `OdraWrapper` class. The class implements all methods defined in `AdWrapper` and `VirtualSchemesManager` interfaces. All internal Mavigator's methods deal only with the mentioned class (accessed via interface type). Below we have enumerated more significant methods and properties (property is a special programmatic construct, introduced in MS C# programming language, which is a some kind of method acting like attribute) with a short comment:

- `void Open(params Object[] par);` Opens the connection to the data source. Depending on particular kind of wrapper, number and/or type of the parameters may vary (that's way object's array is used).
- `Image DefaultIcon;` Returns default icon (which will be used to visualize the class on the surface) for the class.
- `int GetNumberOfObjects(AdUniqueID classUniqueID);` Returns the number of all objects, which belong to the given class (by its unique ID).
- `IList GetAllObjects(AdUniqueID UniqueID);` Returns an instance of the `IList` containing unique IDs of objects being neighborhood of the given object. This is the main method used during extensional navigation.
- `AdObject GetObject(AdUniqueID id);` Returns an object with a given id.

- `IList GetLabels(IList UniqueIDs);` Returns the `IList` of `AdLabel` (pairs `AdUniqueID` and the label of an item identified by this id) based on unique IDs.
- `IList GetObjectsIDs(IList IDs, AdCriterion Criterion);` Returns the `IList` of `AdUniqueID`s for given objects (defined by `IList` of IDs), which satisfy the given criterion. The main method used in filtering objects (intensional navigation).
- `IList GetObjectsIDs(IList IDs, AdAssociation Association);` Returns the `IList` of `AdUniqueID`s for all objects from the target class, which are linked via given association with the given objects (specified by `IList` of IDs). Given objects belongs to the source class of the association; returned objects belong to the target class. This method is used in intensional navigation.
- `AdItem GetLocalItem(AdUniqueID ID);` Returns an item (i.e. object, value of the attribute, class, etc) with given ID.

5.2.3 Working with Wrapper Metadata

The tool, such as Mavigator, could not work without metadata concerning data being processed. The wrapper metadata main object implements `AdMetaData` interface and is returned by a special method of the wrapper. Below we have enumerated `AdMetaData` interface methods:

- `int NumberOfClasses;` Returns the number of classes in the DB schema.
- `IList Classes;` Returns the `IList` of classes (containing instances of the `AdClass`) from the DB schema.
- `int NumberOfAssociations;` Returns the number of associations in the DB schema.
- `IList Associations;` Returns the `IList` of associations (containing instances of the `AdAssociation`) from the DB schema.

Each class from the database is described by a dedicated object of the class implementing `AdClass` interface. The interface has special methods and properties to discover information about classes from the database (below are the most important):

- `String Name;` Returns the name of the class.
- `String Label;` Returns the label (which sometimes could be easier to read for the user) of the class.
- `AdAttribute[] Attributes;` Gets an array, which stores references to objects (instances of the `AdAttribute` interface) defining attributes for this class. `AdAttribute` interface has another set of methods and properties describing particular attribute:
  - `String Name;` Returns the name of an attribute.
  - `String Label;` Returns the label (which sometimes could be easier to read for the user) of an attribute.
  - `bool IsNumber;` Indicates if this attribute is numerical.
  - `bool IsText;` Indicates if this attribute is textual.
  - `AdUniqueID MetaClass;` Returns the id of a class which this attribute belongs to.
- `AdAssociation[] Associations;` Gets an array, which stores references to objects (instances of the `AdAssociation` interface) defining associations of this class. Single association is described by the instance of the `AdAssociation` interface, which has following methods and properties:
  - `AdUniqueID Source;` Returns id of the Source (from) class.
  - `AdUniqueID Target;` Returns id of the Target (to) class.
  - String Name; Returns the name of the association. We always use pair „twin” associations each thus name comply with ODMB pointer’s name.
  - `AdUniqueID ReverseAssociation;` Returns id of a reverse "twin" association. Both of them create bidirectional association (between Source and Target).

- `String QueryDefinition;` Returns a query language definition for an association (executed definition returns objects from the target class). The content of the property depends on kind of data source that we work with, i.e. for the ODRA database it will be an SBQL statement and for a relational database it would be SQL command.

Thanks to the very clear and straightforward API, a programmer, who would like to find out metadata information, should perform the following steps:

- Get instance of the class (`VirtualSchema`) implementing `AdMetaData` interface,
- Get collections containing classes and associations,
- For a particular object defining database class, get arrays defining attributes and associations.
- Work with single objects (being part of the collection or arrays) defining particular metadata items.

5.2.4 Working with Database's Objects

Working with database objects is similar to working with metadata objects. An instance of the database's object (i.e. a particular employee or a product) is represented by an instance of the class which implements `AdObject` interface. Similarly to the metadata's interfaces, this one also has dedicated methods and properties (the most important ones):

- `AdUniqueID MetaClassID;` Gets id of a class the object belongs to.
- `AdLink[] Links;` Gets array which stores references to objects (instances of `AdLink` interface) defining links of the object.
- `AdAttributeValue[] Values;` Gets array of references to the objects (instances of `AdAttributeValue` interface) storing values of attributes of this object.
- `System.String Label;` Returns the value of the label (name) for this object. Returned label could be used to show the "name" of the object

(distinguishing one object from another when the user sees a list of some objects).

Information about a particular object link (instance of an association) is stored by an instance of the class implementing `AdLink` interface. By having the instance of the class, the programmer is able to utilize following methods or properties:

- `AdUniqueID MetaAssociationID;` Returns `AdUniqueID`, which defines association of this link.
- `AdUniqueID SourceID;` Returns id of the Source object's link.
- `AdUniqueID TargetID;` Returns id of the Target object's link.
- `AdUniqueID ReverseLinkID;` Returns the reverse "twin" link. Both of them creates bidirectional link (between Source and Target). The same situation like in case of associations.

Similar situation occurs when the programmer has an instance of the `AdAttributeValue` interface, which stores information about a particular attribute value. This interface defines just two properties:

- `AdUniqueID MetaAttributeID;` Returns information about attribute of the value.
- `System.String Value;` Returns value of the attribute as a string.

Analogical to working with metadata, working with objects' API is also easy. The programmer can run a wrapper method (i.e. `AdObject AdWrapper:: GetObject(AdUniqueID id)`), which returns an instance of the `AdObject` interface. Then can run a dedicated method or use properties to find out values of the attributes or target objects ids (defined by links).

5.2.5 Virtual Schemas

In chapter 3.5 we have described the ideas behind Virtual Schemas (VS) and motivations for introducing them into the Mavigator. In this chapter we would like to focus on implementation details concerning this topic. The first issue, which had to be resolved, was the way of defining a particular Virtual Schema. At the beginning, we have two different ideas:

- The first one assumed introducing some kind of a programming language, which constructs will be used to define virtual schema's items. Then VS module will process a language source code to create classes, attributes, etc.
- The second one was to utilize a special file defining particular items.

In general, both of them allow achieving the same goal. However the first one requires cumbersome actions, such as parsing, interpreting, etc. Thus, finally we have decided to follow the second approach.

A single Virtual Schema is defined in an XML file containing special tags (some of them contain nested "sub-tags" defining "sub-items", i.e.: class and its attribute):

- `<Name>`. Defines a name (visible to the user) of the VS,
- `<Description>`. Contains description for the VS,
- `<Class>`. Defines a class and contains additional, nested sub-tags:
  - `<Name>`. The name of the class,
  - `<Attribute>`. Describes an attribute and contains additional, nested sub-tags:
    - `<Name>`. The name of the attribute,
    - `<Type>`. The type of the attribute. Because the wrapper is dedicated to the ODRA database, the type belongs to the ODRA namespace, i.e.: `DataStore.STRING_TYPE`,
    - `<SBQLDefinition>`. This tag's value contains an SBQL query language statement, which returns the value of the attribute. It could be any valid statement (including methods, over attributes, calculations, etc.).
- `<Association>`. Defines bi-directional association between two classes. We have decided to place (in file) association's definition outside the class definition and after all classes' definitions. Thanks to the solution, we avoid situation, when the association refers to the class, which definition has not been read from the file yet. Below we enumerated all tags used in the association definition:

- `<Name>`. The name of the association's role (visible to the user),
- `<ReverseName>`. The second name (in the opposite direction) of the association's role (visible to the user),
- `<SourceClassName>`. The name of the source class for the association,
- `<TargetClassName>`. The name of the target class for the association,
- `<SourceSBQLDefinition>`. This tag contains an SBQL query language statement, which defines the source association. It could be any valid statement (including methods, over attributes, calculations, views, etc.); however the semantic of the associations requires returning of the object's ids from the target class. Thus, usually it will be some kind of path expression,
- `<TargetSBQLDefinition>`. This tag contains an SBQL query language statement, which defines the target association (the second one, reverse). The same remarks apply like in the previous case.

Example 16 shows a definition of the simple Virtual Schema containing only two classes and one association.

Virtual schemas are widely utilized in the entire wrapper. Almost every call to the data source (through the wrapper) is mapped from the database view defined in VS to particular, physical data's items stored in the data source. Following sub-chapters give implementation details for particular actions.

Before we start, we would like to explain topic according UML's sequence diagrams, which will be used to illustrate particular issues. On the diagram there is an entity called actor, who starts the whole process. Because Mavigator's is an application with the graphical user interface, all actors' actions are performed through some widgets. Thus, on all the diagrams, there would be an actor, calling some methods from the widgets' classes and then, there would be significant callings of the described methods. Therefore, we have decided to join virtually (just for the diagrams purposes) all particular widgets' classes into one called just GUI. Moreover, to improve

readability of the diagrams, we have decided to skip an actor and mark all actions (where appropriate) as started by the GUI object.

```
<VirtualScheme>
  <Name>Simple Virtual Scheme</Name>
  <Description>Simple virtual schema.</Description>
  <Class>
    <Name>Orders</Name>
    <Attribute>
      <Name>OrderDate</Name>
      <Type>DataStore.STRING_TYPE</Type>
      <SBQLDefinition>OrderDate</SBQLDefinition>
    </Attribute>
  </Class>
  <Class>
    <Name>Shippers</Name>
    <Attribute>
      <Name>ShipperID</Name>
      <Type>DataStore.INTEGER_TYPE</Type>
      <SBQLDefinition>ShipperID</SBQLDefinition>
    </Attribute>
  </Class>
  <Association>
    <Name>ShipVia</Name>
    <ReverseName>Ships</ReverseName>
    <SourceClassName>Orders</SourceClassName>
    <TargetClassName>Shippers</TargetClassName>
    <SourceSBQLDefinition>Orders.ShipVia.Shippers;</SourceSBQLDefinition>
    <TargetSBQLDefinition>Shippers.Ships.Orders;</TargetSBQLDefinition>
  </Association>
</VirtualScheme>
```

Example 16. XML file defining simple virtual schema.

5.2.5.1 Opening a Data Source

As mentioned previously, opening a particular database is done via `void Open(params Object[] par)` method, declared in the `AdWrapper` interface

and implemented by the `OdraWrapper` class. Figure 71 shows sequence diagram illustrating the actions taken inside the method:

- At the beginning the number of arguments is checked (call number 2). If the number is different then two, an exception is raised,
- Then, in call number 3, instance of the `Hashtable` is created, which is used as a some kind of cache memory (see chapter 5.2.5.2),

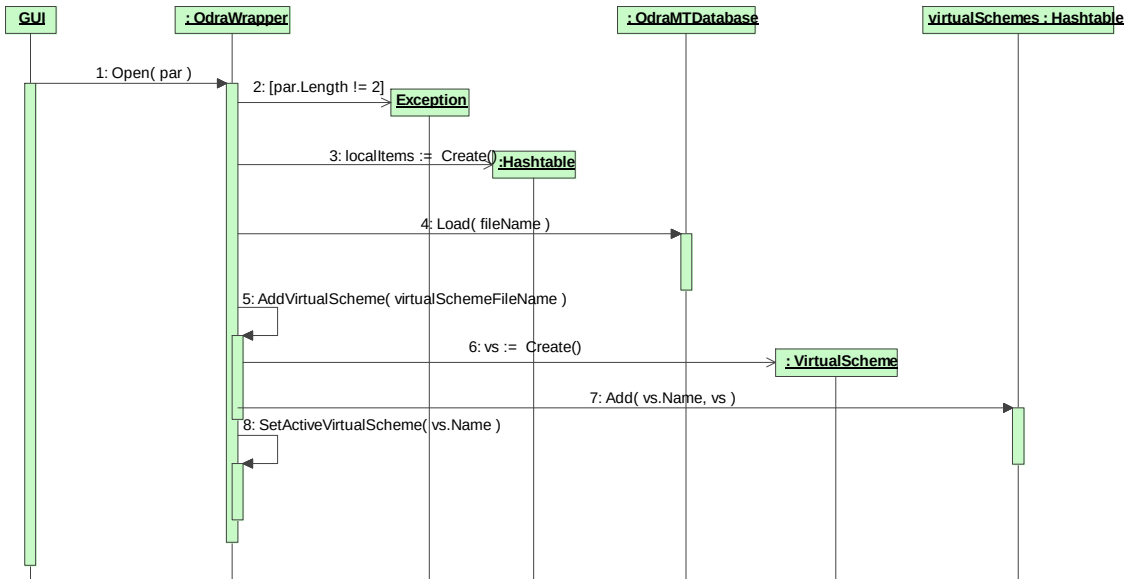

Figure 71. Sequence diagram for the `OdraWrapper::Open` method.

- Loading data from the file into the database memory – call number 4,
- Next, call number 5 adds virtual schema,
  - Creating a new instance of the `VirtualSchema` (call number 6; more information could be in found chapter 5.2.5.3) and adding them to the `Hashtable` storing all loaded virtual schemas (call number 7).
  - And finally setting the newly created virtual schema as an active one (call number 8).

5.2.5.2 Using Cache

Performance is a key issue during working with all kinds of database systems and applications working with them. One of the common techniques is using a cache

memory, which stores items downloaded from the repository. Thus referencing to the particular item (link, attribute's value, etc) or meta item (class, association, etc) does not require recreating them on the application side. We have followed this approach in our wrapper too. All items downloaded from the database are stored in a special map, where the key is a unique id of the item (instance of the `AdUniqueID`) and a value is a reference to the object. We have decided to use instance of the `Hashtable` class (named `localItems`), which supports a good performance.

There are a few simple methods, which are used to manage the cache memory, namely:

- `AdItem GetLocalItem(AdUniqueID ID);` The method returns an item (object, link, attribute's value) or meta item (class, association, attribute) with a given id.
- `int LocalItemSize;` The property simply gets the number of items stored in the cache.
- `void ClearUserObjectForLocalObjects();` The method finds all objects, which are stored in the cache and clear their user's object. This is necessary in some cases (see further chapters).
- `void ClearUserObjectForLocalAttributeValues();` Similar to the above method, but clears for the attributes' values.

One could be curious why we do not have a method to clear the cache (remove all stored objects). In language like C#, the cleaning job is done by the garbage collector, which will clear the memory, when there will be no references to the cache collection.

5.2.5.3 Loading Virtual Schema

All task related to the creating virtual schema are performed in the constructor of the `VirtualScheme` class. Figure 72 illustrates the whole process:

- Call number 1 creates the object (also shown on Figure 71, page 105) and all actions are started from the constructor,

- Two instances of the `ArrayList` class are created. They store objects of the `AdOdraClass` (classes, call number 2) and `AdOdraAssociation` (associations, call number 3),
- Then `Hashtable fromUniqueIDToItem` is created, mapping from items' ids to the items' instances. This map is required by the method `AdWrapper::GetLocalItem` presented in the chapter 5.2.5.2,

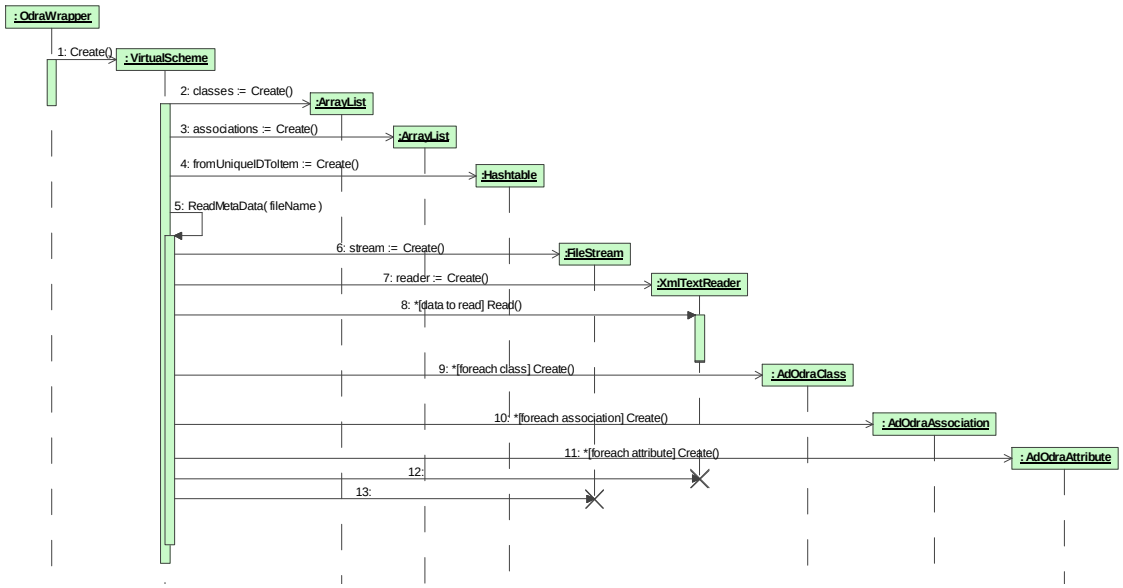

Figure 72. Sequence diagram for creating Virtual Schema.

- Call number 5 starts the `ReadMetaData` method, which do the following:
  - Creates two objects (`FileStream`, `XmlTextReader`) needed to read virtual schema configuration from the XML file,
  - Reads all XML data and creates instances of the `AdOdraClass` (information about a class), `AdOdraAttribute` (information about an attribute) and `AdOdraAssociation` (information about an association).

More about working with metadata could be found in Chapter 5.2.3.

5.2.5.4 Accessing an Object

Objects from a data source are represented by the `AdOdraObject` class, which implements `AdObject` interface. They could be accessed using `AdObject GetObject(AdUniqueID id)` method, which body is presented on the sequence diagram from the Figure 73:

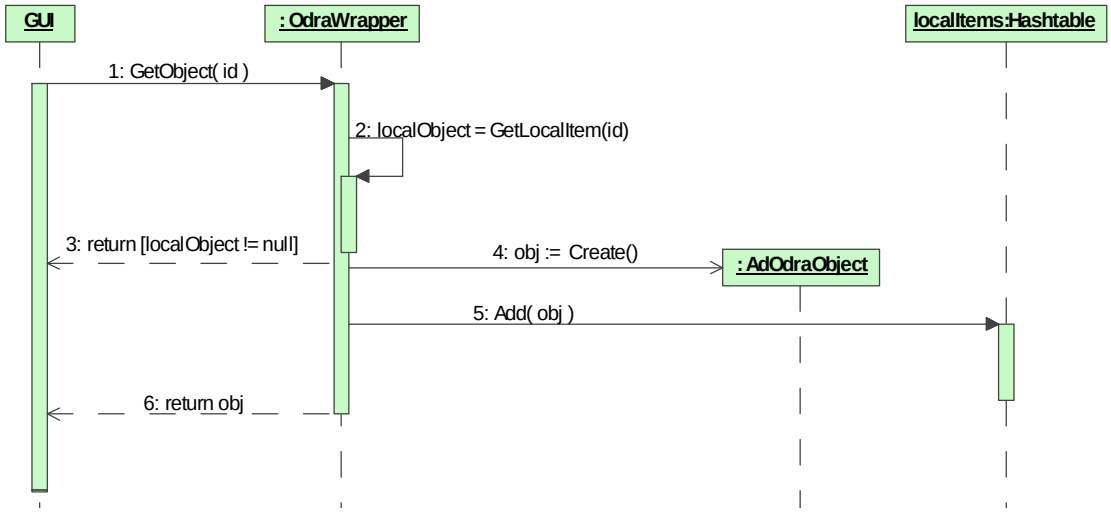

Figure 73. Sequence diagram for the `OdraWrapper::GetObject(Id)` method.

- The first thing, which should be performed is to check if required object is in the cache memory (see chapter 5.2.5.2) – call 2.
- If the method returns value different then `null`, which means a valid object, the object is returned – call 3.
- Otherwise the object's instance is created – call 4 (see further in this chapter),
- Newly created object is added to the cache memory – call number 5 and returned in call number 6.

As always in this wrapper, the entire process of creation and initialization with proper values is performed in the constructor, which is illustrated on Figure 74 and Figure 75 (sequence diagram has been split into two because of the readability issues):

- At the beginning two instances of the `ArrayList` class are created. They store attributes' values – (call a1) and links (call a2).

- Then in call a3 the id of the object's class is retrieved and, in call a4, the object's class itself,
- Call a5 creates OID (special object's ids used in ODRA database), which will be used to retrieve "original" ODRA object,

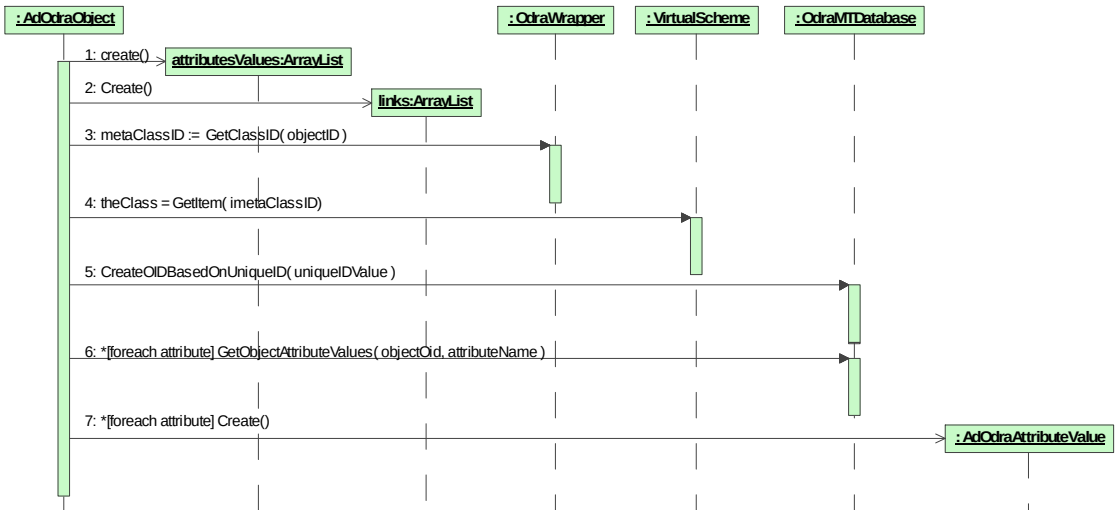

Figure 74. Sequence diagram for creating an object – part (a).

- In call a6 we iterate through all attributes, and for each we retrieve all its values (because in ODRA all attributes could have many values) and create corresponding `AdOdraAttributeValue` instances (call a7),
- Analogical steps are performed for links (calls b1 to b6).

5.2.5.5 Downloading object's Neighborhood

The process of download the neighborhood is widely utilized during extensional navigation (implementation details about the topic could be found in chapter 5.4). In this chapter we would like to discuss only wrapper's part of the process. The wrapper's method, which allows downloading the neighborhood is `IList GetAllObjects(AdUniqueID UniqueID)`. Its content (as a sequence diagram) is shown on Figure 76 and could be summarized as follows:

- At the beginning (call number 2) the instance of the `ArrayList` is created. This object will store ids of the objects composing the neighborhood and id of the starting object,

- Then, starting object is downloaded (as an instance of the `AdOdraObject` class) – call number 3,

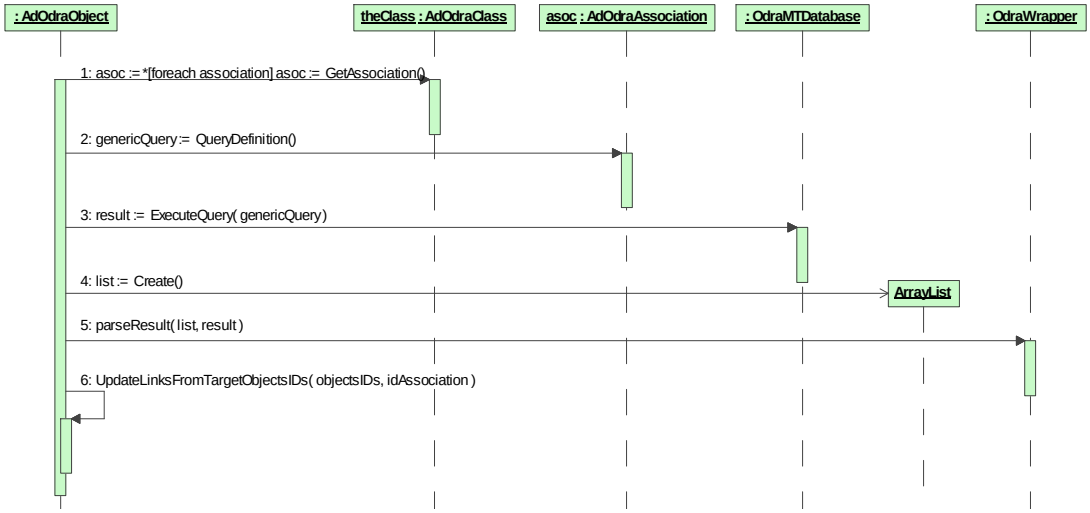

Figure 75. Sequence diagram for creating an object – part (b).

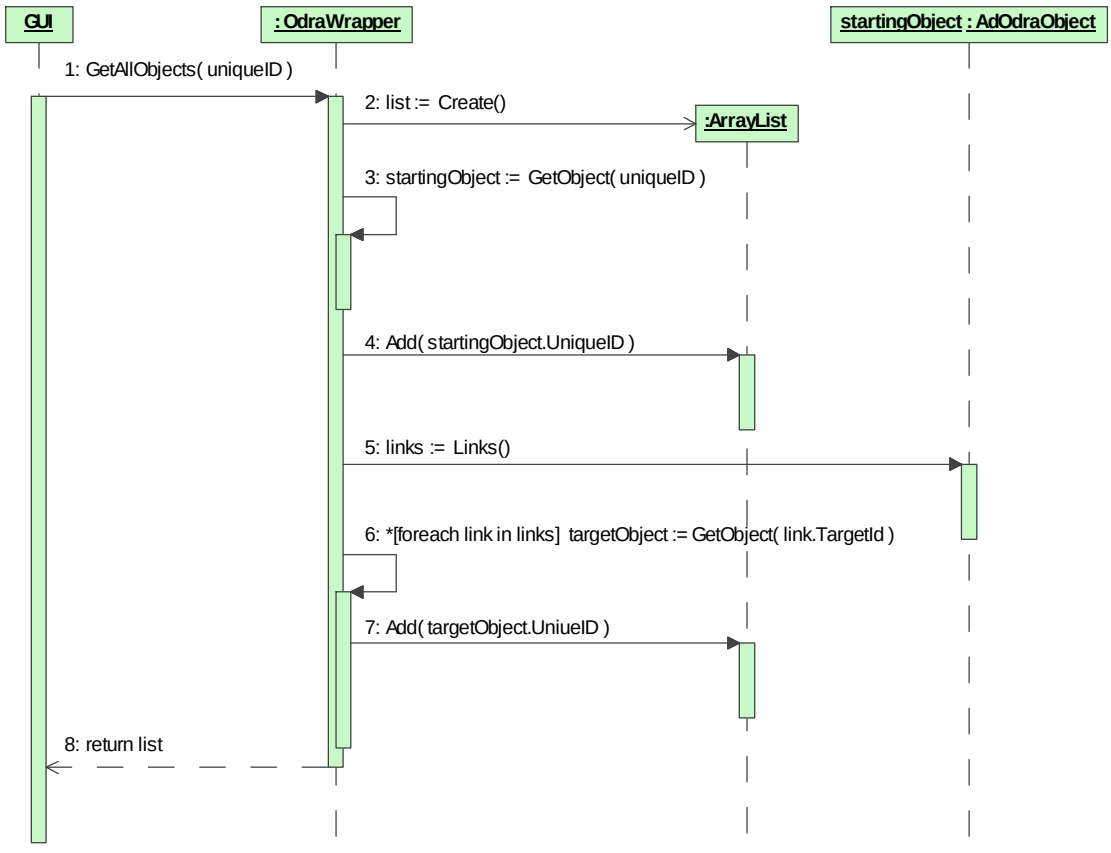

Figure 76. Sequence diagram for downloading object's neighborhood.

- Identifier of the starting object is added to the previously created collection – call number 4,
- For the starting object, collection storing its all links (instances of the `AdOdraLink` class) is accessed in call 5,
- Each object being link's target, is downloaded (call 6) and added to the results' collection (call 7).
- Finally, in call number 8, the collection is returned.

5.2.5.6 Navigation via Association's Role

Navigation via association's role is utilized in the intensional navigation session. From the wrapper's point of view, the `IList GetObjectsIDs(IList IDs, AdAssociation Association)` method is crucial. Figure 77 illustrates the actions performed in the method:

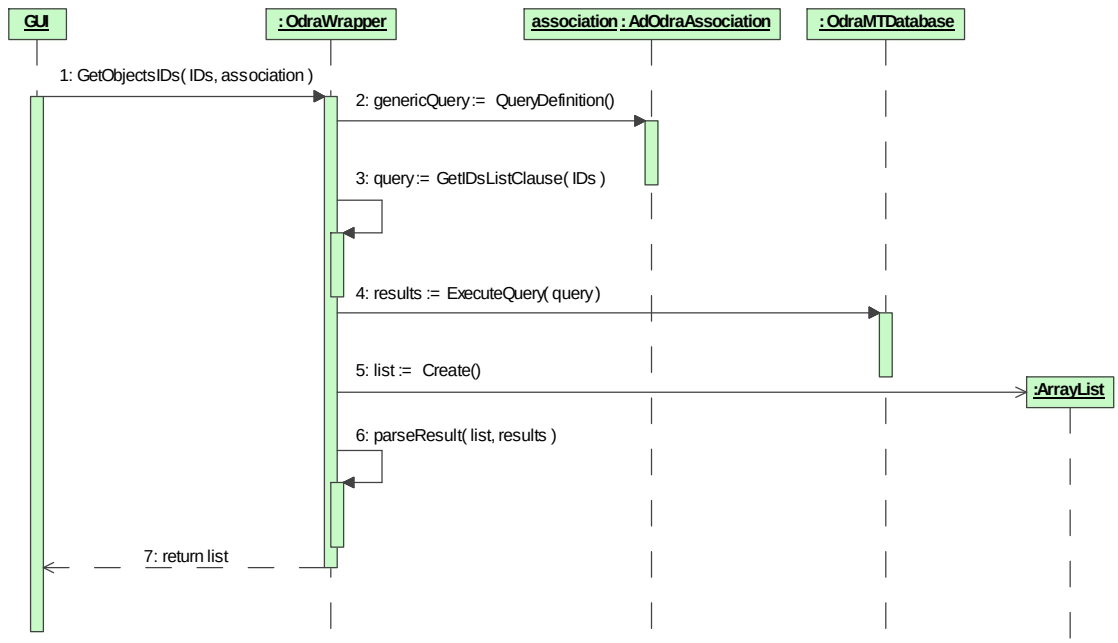

Figure 77. Sequence diagram for navigating via association.

- First the query language statement for the given association is read (call number 2),

- Then query is modified in such a way to navigate only from given (passed to the method) objects (call number 3),
- In call 4 the query is executed,
- Call number 5 creates an `ArrayList`, which will store the ids of the objects being the result of the navigation,
- Returned result (from call 4) is parsed, converted (from ODRA native results to the `AbstractDatabase` format) and stored on the list.
- And call 7 returns the collection containing the result of the navigation.

5.2.5.7 Filtering Objects

Filtering objects is utilized in the intensional navigation as a one of the ways of marking objects. Wrapper has two methods performing the task:

- `IList GetObjectsIDs(AdCriterion Criterion);`
- `IList GetObjectsIDs(IList IDs, AdCriterion Criterion);`

The first one simply calls the second one with the first parameter equals to `null`. Thus we are going to discuss only the second one. Its content is presented on the sequence diagram shown on Figure 78:

- In call number 2, the method checks if collection of objects' ids is not null. If it is so, query containing information about objects is formulated,
- If passed `Criterion` type is different than `AdCriterionFilter`, an exception is thrown. This is caused by the fact, that this prototype implementation only works with filtering (call number 3),
- Calls 4 and 5 get object describing the class being filtered,
- In call 6 query (filtering) is executed,
- Call number 7 creates an `ArrayList`, which will store the ids of the objects being the result of the filtering,

- Returned result (from call 6) is parsed (call 8), converted (from ODRA native results to the `AbstractDatabase` format) and stored on the list,
- And call 9 returns the collection containing the result of the filtering.

Figure 78. Sequence diagram for filtering objects.

5.2.5.8 Working with Labels

Sometimes in Mavigator we have to deal with objects' labels. It happens often during working with baskets or the list of objects. Thus there must be a method, which for given objects' ids, return their labels (actually labels and ids). And this actually what the `IList GetLabels(IList UniqueIDs)` method do: returns a collection containing instances of the `AdOdraLabel` class. The whole process is illustrated on the sequence diagram from Figure 79:

- First the `ArrayList` is created for storing the list containing pairs: object's id and label (stored in instance `AdOdraLabel`) – call 2,
- Then for each object's id from the passed list – call 3:
  - Get class's id for an object with given id (call 4),

- Get the class object with the above id (call 5),
- Find attribute acting as object's label – call 6,
- Get value of the attribute in call 7 (simplified on diagram- actually the array of values is processed),
- Create an instance of the AdLabel (call 8) and add it to the list (call 9).

- And finally, after processing all objects' ids, the list is returned in call 10.

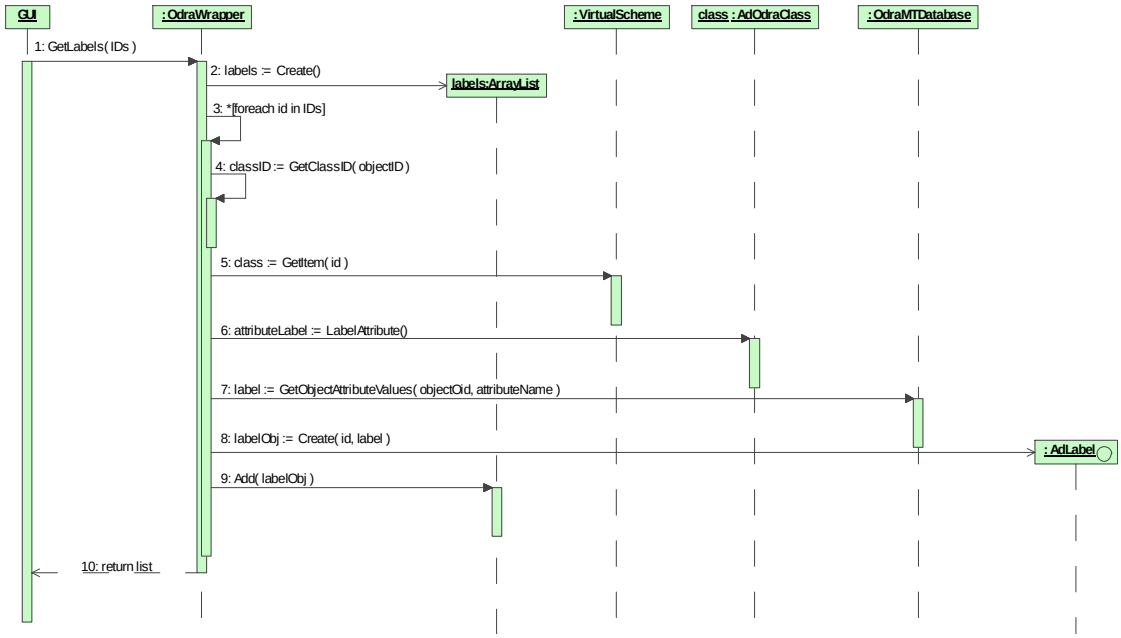

Figure 79. Sequence diagram for creating labels for objects.

5.2.5.9 Dealing with configuration

Mavigator's wrapper works with two kinds of data, which could be recognized as a configuration:

- Virtual Schemas, because they influence the way how user see database schema graph,
- General configuration file, dedicated to the database.

Because Virtual Schemas' configuration has been already discussed we will concentrate only on the second case.

Wrapper's configuration is stored in an XML file containing different sections, related to the particular topics:

- `Objects.Labels`. For each objects' class, defines name of the attribute, which values will be used as a object's labels,
- `Objects.Images`. For each objects' class, defines the file name storing a graphics (icon), to visualize the object (i.e. during extensional navigation),
- `Classes.Locations`. For each class, defines its location (in terms of coordination's: x, y) on the surface in the intensional navigation window,
- `Classes.Images`. For each class defines the file name storing a graphics (icon), to visualize the class (in intensional navigation). The file could be different than file used to visualize objects of the same class.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Configuration file for the Odra AdWrapper
DatabaseName: Northwind -->
<profile>
  <section name="Objects.Labels">
    <entry name="Employees">LastName</entry>
    <entry name="Customers">CompanyName</entry>
  </section>
  <section name="Objects.Images">
    <entry name="Employees">ClassesIcons\MovieDirector.png</entry>
    <entry name="Customers">ClassesIcons\Person.png</entry>
  </section>
  <section name="Classes.Locations">
    <entry name="Employees">550;400</entry>
    <entry name="Customers">200;700</entry>
  </section>
  <section name="Classes.Images">
    <entry name="Employees">ClassesIcons\MovieDirector.png</entry>
    <entry name="Customers">ClassesIcons\Person.png</entry>
  </section>
</profile>
```

Example 17. Sample configuration file.

Example 17 shows sample XML file containing configuration for the simplified (cut to only two classes) Northwind database.

5.3 Intensional Navigation

Before we go into details, we notice that **a**lmost all calls to the "business" Mavigator's methods (like the starting new intensional navigation: `startIntensionalNavigation()`) are the results of some GUI interaction. Thus, they are done via dedicated handler methods (linked to the particular widgets, and triggered on specified user's actions. i.e. click) i.e. `mnuViewIntensional Navigation_Click`. However, because of clarity, they are skipped in descriptions and diagrams.

The next sub-chapters discuss particular topics related to the implementation of the intensional navigation.

5.3.1 Starting a New Session

When the user starts intensional navigation session using appropriate command of the Mavigator's main menu, the `startIntensionalNavigation` method from the `MainFrame` class is run.

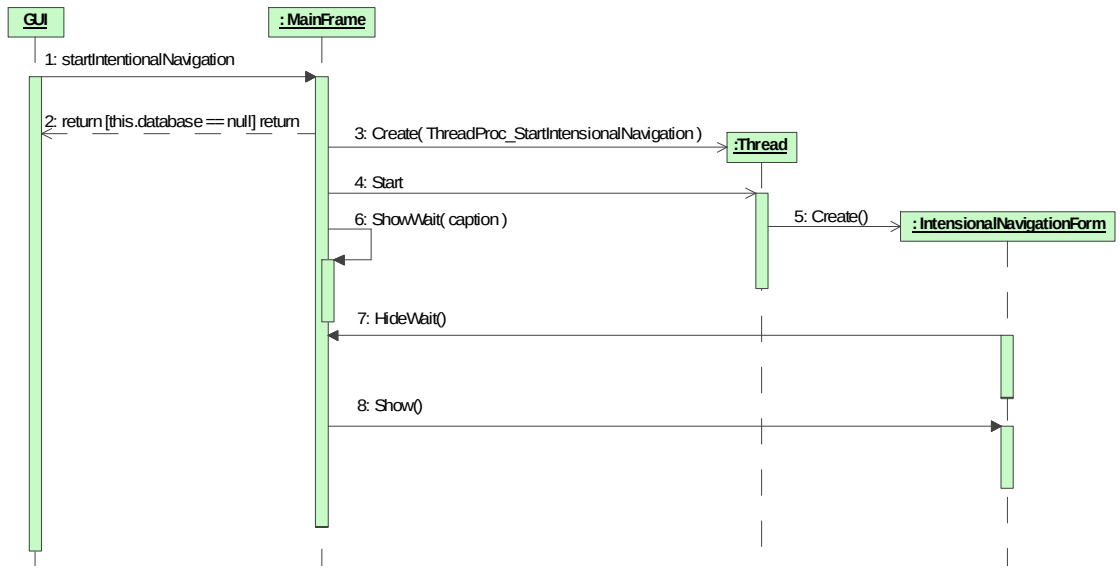

Figure 80. Sequence diagram for starting a new intensional navigation session – an overview.

The process's explanation has been split (because of the diagram's size) into three parts. The first one, an overview, illustrated by the sequence diagram shown on Figure 80, consists of the following elements:

- Checking if data source has not been opened. In that case, the special message is shown (not presented on the diagram) and starting is interrupted – call number 2,
- Creating a new thread with a method passed as a parameter – call number 3. In fact, the thread's utilization looks little bit more complicated. However we have skipped some items, used in code, but not shown on the diagram.
- Starting created thread – call 4,
- Showing "wait" window to the user (call 6). In the mean time, the started thread creates (call 5) a new instance of the `IntensionalNavigationForm` class – see further,
- When creating is done, the method `HideWait()` is called (message number 7),
- Finally, after unblocking modal "waiting" window, the `Show()` method is run – call number 8.

The second explanation's part discuss creating (in a separate thread) and initialization instance of the `IntensionalNavigationForm` class. The sequence diagram, shown on Figure 81, helps to understand the process:

- Call 2 among creating ordinary window's widgets (not shown) creates (call 3) instance of the `GraphPanel` class, which is used to visualize database schema graph,
- In call 4 some graph's properties are set (shown only one of them) and in call 5 (shown only one) handler methods are assigned,
- Call 6 uses global class method to load icons for the toolbar,
- Call 7 run significant method, which fills the graph with edges and vertices (see further),

- And the last call (8) creates instance of the `UndoManager` class to handle undo (back) and redo (forward) operations.

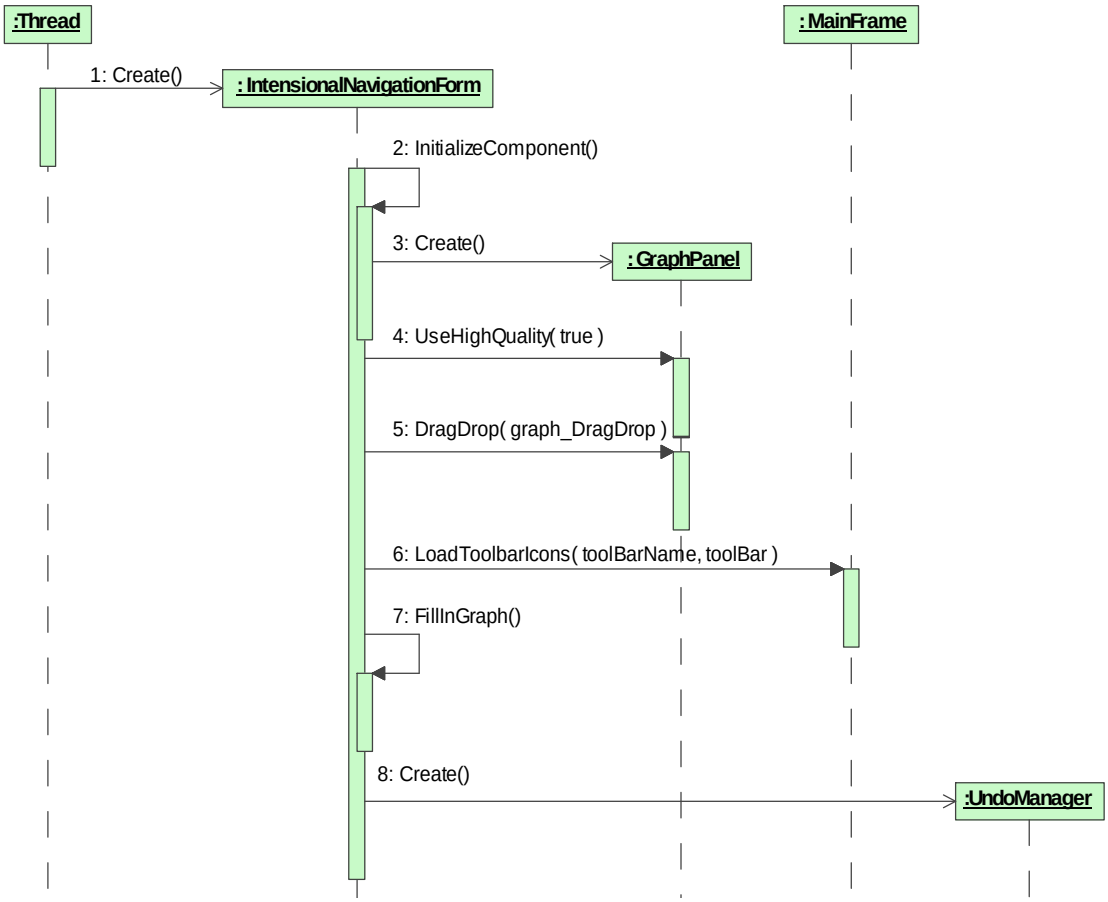

Figure 81. Sequence diagram for creating instance of the `IntensionalNavigationForm` class.

And the last part discusses the `FillGraph()` method, which fills the graph with classes and associations (and roles). Corresponding sequence diagram (simplified) is shown on Figure 82:

- At the beginning, three map collections are created (calls: 2, 3, 4),
- Then for each class in the data source:
  - Add vertex (call number 6) as a visualization of the class,
  - Remember number of objects for this class in the map (call 7),

- Remember mapping from class's id to the corresponding vertex (call number 8),

- For each association (simplified description):
  - Add edge to the graph (call 10),
  - Remember mapping from association's id to the edge,
  - Add role's label for the association.

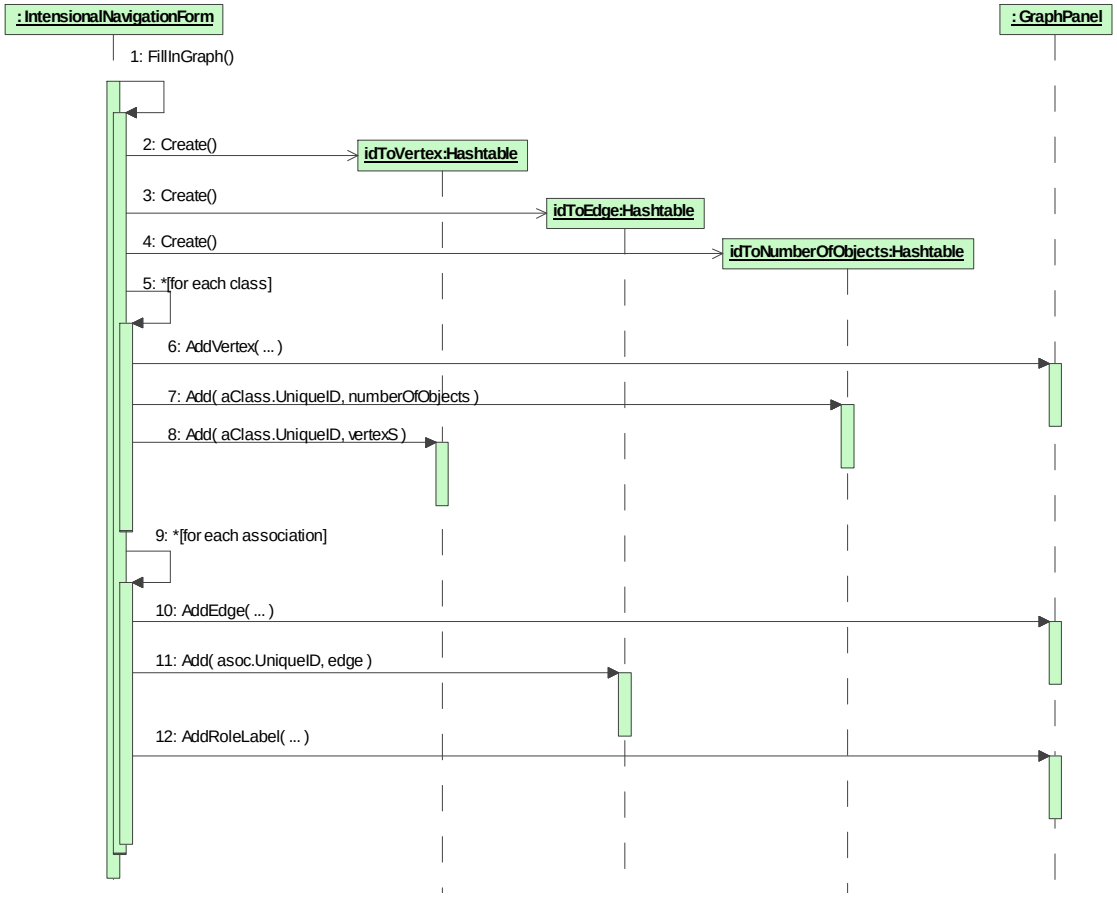

Figure 82. Sequence diagram illustrating `FillInGraph()` method.

5.3.2 Filtering Objects

Filtering objects is used, as a one of the ways, to marking objects in intensional navigation. It is started from the context menu (Figure 30, page 56), which means creating instance of the `ContextMenu` class, showing it and catching user's choice.

As a result the `FilterMarkedObjects()` method is started. Their interior (in a simplified form) is shown using sequence diagram on Figure 83:

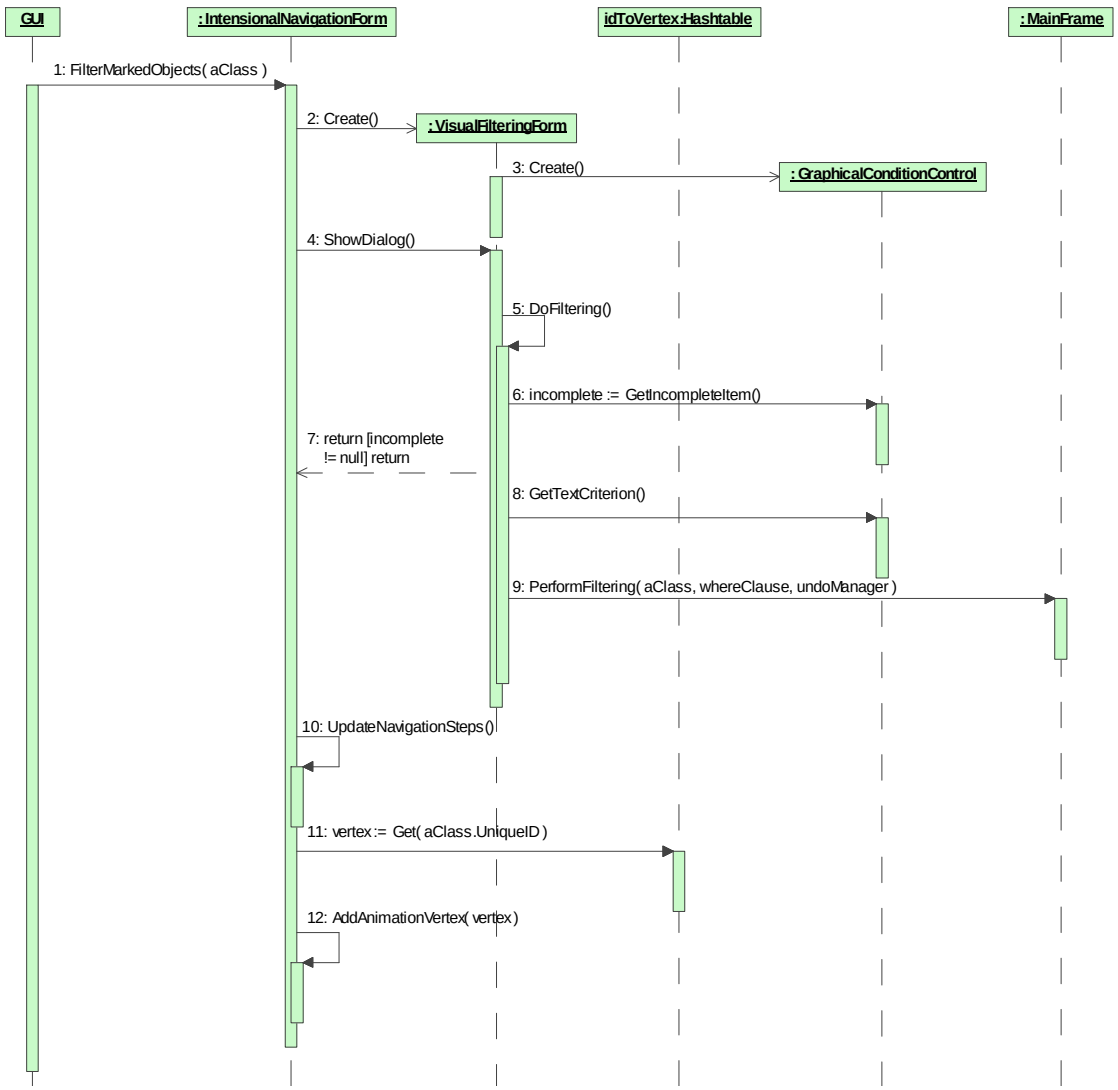

Figure 83. Sequence diagram for filtering objects.

- Call 2 creates window of the `VisualFilteringForm` class, which is used to graphically (see 4.3.1) formulate the filtering criterion.
- In call 3 the instance of the `GraphicalConditionControl` class is created,
- Call 4 shows the window as a modal dialog. User is able to "draw" a criterion and then clicks the *Filtering* button, which runs `DoFiltering()` method (call 5),

- Call 6 checks if criterion is complete. Otherwise returns (call number 7),
- Call 8 gets filtering criterion, visually formulated by the user,
- Call 9 starts filtering asynchronously (using threads, not shown on the diagram),

- In call 10, history of marked objects' activities is updated (see chapter 5.3.6),
- Finally, calls 11 and 12 handle animation of the filtered class (graph's vertex).

5.3.3 Navigating

Navigating is started when a user clicks on the association's role. As a result, `Navigate(AdAssociation association, Point point)` method is started, which shows the context menu and run `CreateAndRunNaviagtionThread()` method. Their operations (simplified version) are illustrated on the sequence diagram shown on Figure 84:

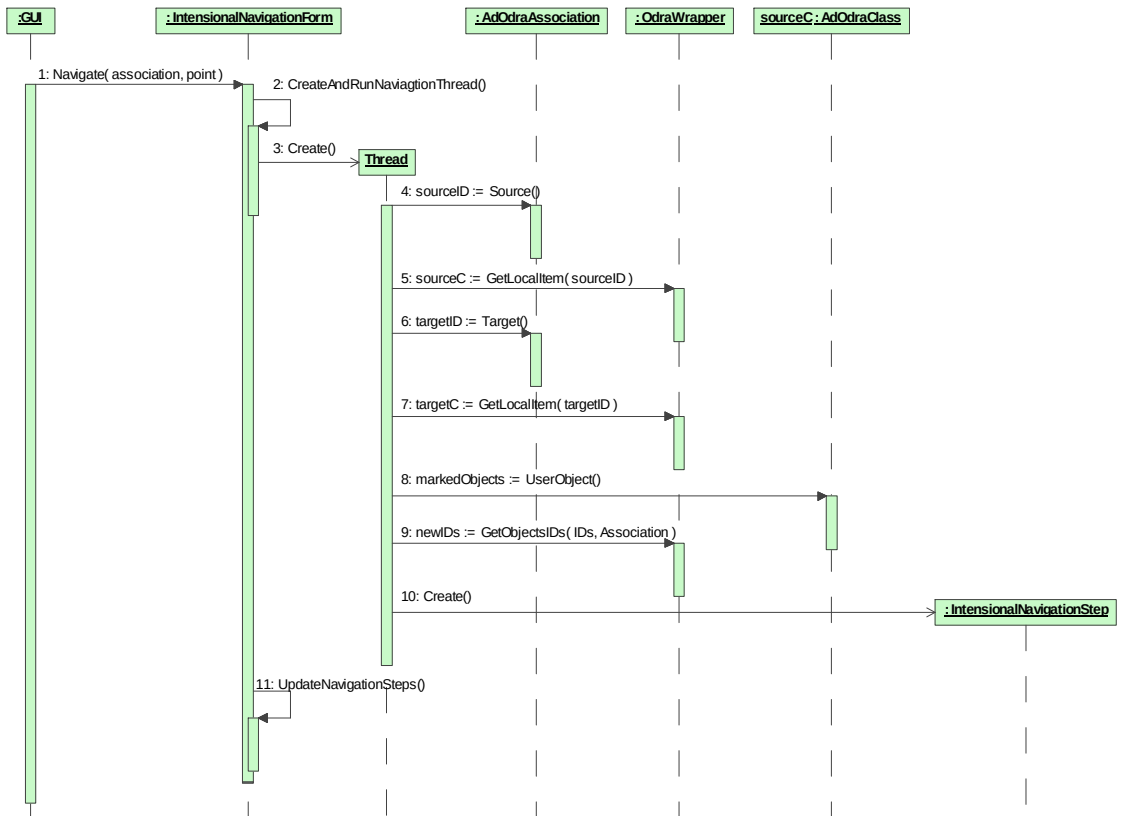

Figure 84. Sequence diagram illustrating navigating from class to class via association's role.

- Call 1 displays context menu with navigation's options and handles user's input (also deals with the progress window – not shown on the diagram),
- Call 2 starts a new thread performing navigating:
  - Calls 4 - 7 gets instances of classes describing source and target classes of the navigation,
  - Call 8 retrieves a collection (stored as a user's object) holding class's set of marked objects,
  - Call 9 refers to the data wrapper and retrieves ids of the objects being the result of navigation,
  - Call 10 creates an object, which is used to performing undo of the navigation,
  - And call 11 updates navigation steps (see chapter 5.3.6).

5.3.4 Showing Objects

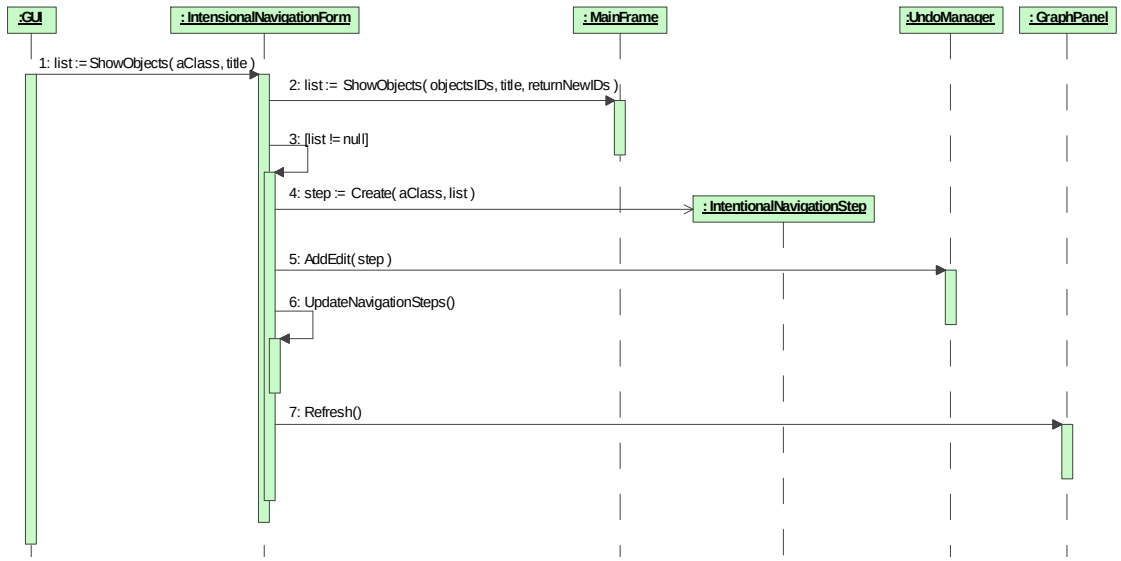

Figure 85. Sequence diagram for showing objects – part 1.

A list of marked objects could be shown during an intensional navigation session. The appropriate command is started from the marked objects' context menu and results in running the `void ShowObjects(AdClass aClass, String title)`

method from the `IntensionalNavigationForm` class. This operation performs only two general steps (see Figure 85):

- starts a more general version (call 2) from the `MainFrame` class - `IList ShowObjects(IList objectsIDs, String title, bool returnNewIDs)`,
- checks if the shown list has been modified (call 3). If so then add instance of the `IntensionalNavigationStep` class (call 4), which records the previous state of the list. Then updates the steps list (call 5, 6) and refreshes the graph (call 7).

The main part of showing objects is performed inside `IList ShowObjects(IList objectsIDs, String title, bool returnNewIDs)` method of the `MainFrame` class. Its functioning has been illustrated on the sequence diagram shown on Figure 86:

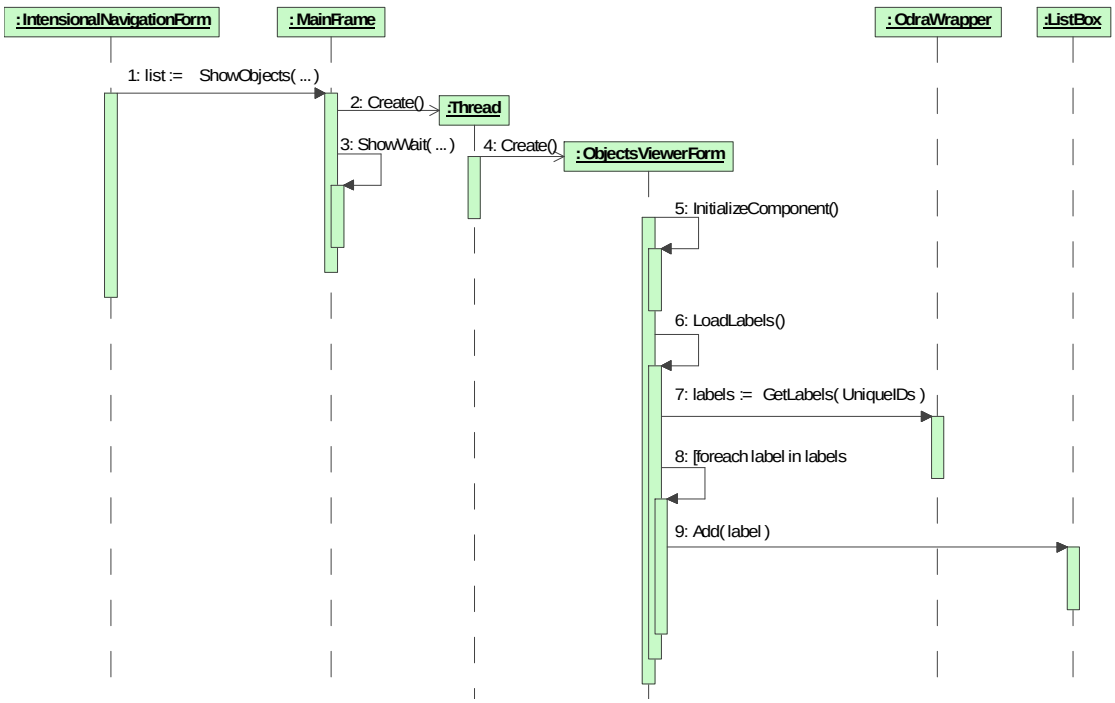

Figure 86. Sequence diagram for showing objects – part 2

- call 2 shows simplified version of creating a new thread, which is responsible for the whole process,

- call 3 shows progress window and “locks” the main thread until the new one ends,
- call 4 creates a window for showing objects’ labels,
- call 5 initializes widgets (also adds listener for user’s actions),
- call 6 loads labels from the wrapper and adds them to the appropriate widget (shown on calls 7 – 9).

5.3.5 Handling Drag&Drop

The Drag and Drop metaphor is widely used in nowadays application. In Mavigator and intensional navigation particularly, it is used to perform two main tasks:

- Dropping object from the basket or extensional navigation,
- Dragging class’s visualization (icon) and drop onto the basket.

The Mavigator’s prototype utilizes the native C# mechanism, which simplifies the whole task. Objects, which are dragged among different Mavigator’s metaphors, are stored inside two kinds of instances:

- `TransporterAdClass` class, which is used during transfer to the basket, and stores instance of the `AdClass`,
- `TransporterAdUniqueIDs` class, which is utilized more widely: in baskets, extensional navigation and intensional navigation. This instance stores the `IList` with ids of the objects being dragged.

The general idea is as follows:

- Add an appropriate handler method to the control (widget), which catches particular kinds of D&D process (start dragging, dropping, dragging over, etc), i.e. the `void graph_DragDrop(object sender, DragEventArgs e)` method catches dropping the item onto the intensional navigation surface,
- Inside the dedicated handler method, the programmer has to:
  - Take out a transferred item from the instance of the `DragEventArgs` class,

- Perform appropriate actions, according to the operation's semantics,
- Set an appropriate effect (special property of the `DragEventArgs` class) on the operation.

5.3.6 Dealing with Back/Forward

Operations related to the marked objects are stored on a special list as so-called edits. The main concept is based on the Java's `UndoManager` class. All actions are supported by the couple of classes:

- `UndoManager` (which implements `IUndoManager` interface) manages and stores edits. More important methods:
  - `void AddEdit(IEditItem edit)` – adds single edit to the list,
  - `void Undo()` - Performs an undo operation for the current edit,
  - `void Redo()` - Performs a redo operation for the current edit,
  - `void UndoTo(IEditItem edit)` - Undoes all changes from the current edit to the given edit,
  - `void RedoTo(IEditItem edit)` - Redoes all changes from the current edit to the given edit,
- `IntensionalNavigationStep` (which implements `IEditItem` interface), stores and manages single edit (navigation step). More important methods:
  - `bool CanRedo()` - Returns true if a redo operation would be successful now, false otherwise,
  - `bool CanUndo()` - Returns true if a undo operation would be successful now, false otherwise,
  - `String GetPresentationName()` - Provides a localized, human readable description of this edit,
  - `void Undo()` - Undoes the edit that was made,
  - `void Redo()` - Re-applies the edit, assuming that it has been undone.

- `override String ToString()` – overridden version of the native C# method. Allows presenting correct labels in C# widgets.

All the above methods are self-explained thus we are not going to discuss them. A short overview of the general idea is as follows:

- all edits are stored by the `UndoManager`,
- each single edit stores two version of data:
  - after the change,
  - before the change,
- when a user would like to perform the undo operation, `UndoManager` calls the `Undo` method for the current edit. As the result, the previous version of the data is set,
- opposing actions are taken during requesting redo operation.

5.4 Extensional Navigation

From the implementation point of view, extensional navigation is resource demanding. It is mainly caused by the potentially big number of objects to work with. However, due to the nature of this research, we avoid complicated optimization techniques, which could have destructing impact on the clarity of the prototype design. The following sub-chapters discuss particular topics.

5.4.1 Starting Navigation

Extensional navigation could be started by the appropriate command from the object context menu. However, the menu is accessible to the user, in many Mavigator's metaphors, including baskets, object operations, etc. Still, all actions refer to the single general method (from the `MainFrame` class): `void StartExtensionalNavigation(AdUniqueID idObject)`. The method works as follow:

- creates an instance of the `ExtensionalNavigationForm` class,
- creates an empty `ArrayList` and adds a starting object,

- shows the form and starts the `void FillGraph(IList listStartingObjectIDs)` method, which downloads the starting object neighbourhood.

5.4.2 Downloading Neighbourhood

Roughly speaking, downloading object's neighbourhood means running the `void FillGraph(IList listStartingObjectIDs)` method with a starting object's id passed as a parameter. Operations performed by the method are illustrated using sequence diagram shown on Figure 87:

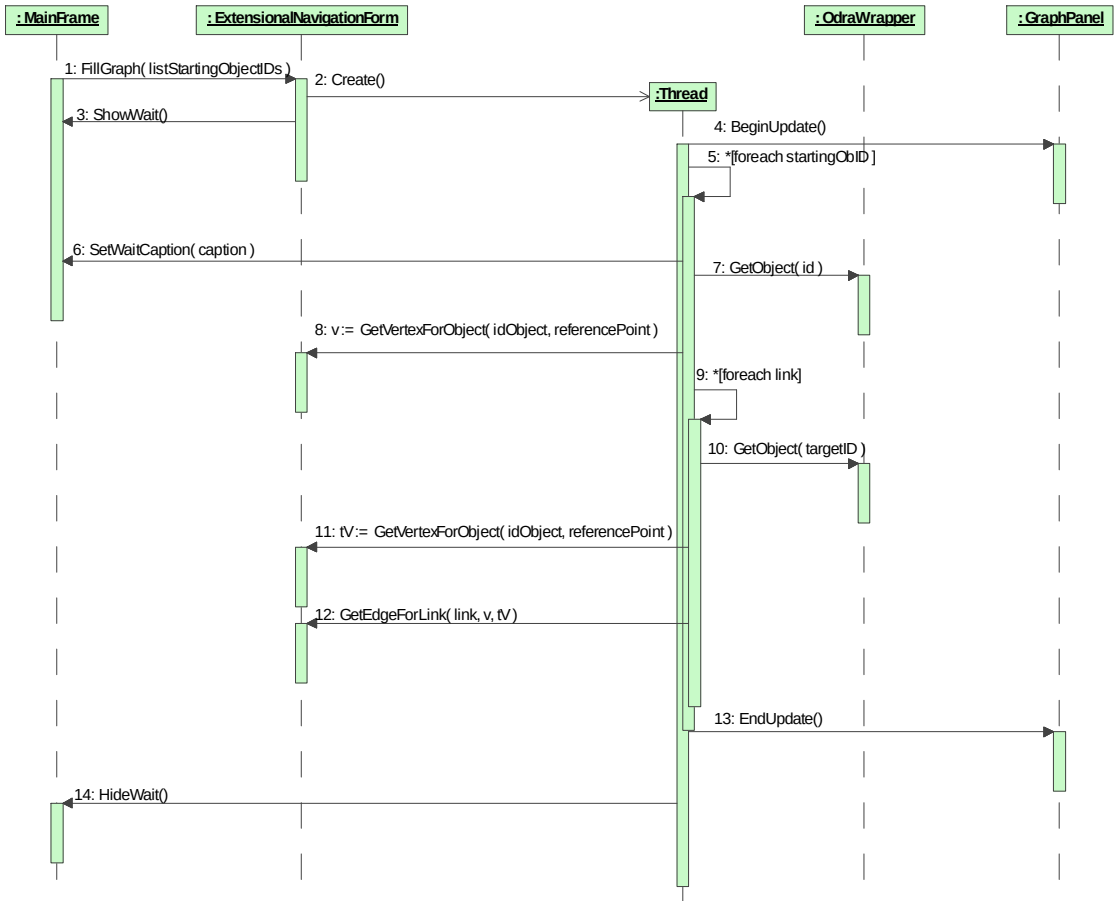

Figure 87. Sequence diagram for downloading object's neighborhood (extensional navigation)

- Call 2 creates a new thread (simplified version), which is responsible for the operation,
- Call 3 "locks" GUI and shows waiting window,

- Call 4 notifies the graph panel that there will be changes to the graph,
- Call 5 process each object's id separately:
    - Call 6 changes the waiting caption,
    - Call 7 gets the "source" object,
    - Call 8 gets vertex for the source object (or creates a new one if required). Each extensional navigation session has its own set of visualized objects. The same object, even referenced from different places is always represented by the same icon on the graph
    - Call 9 processes each link separately:
        - Call 10 gets object (instance of the `AdObject` class) for the target object's id of the processed link,
        - Call 11 retrieves a vertex for the above object,
        - Call 12 returns (or creates if necessary) an edge for the source and target vertex,
    - Call 13 notifies the graph about ending the changes,
- Finally, call 14 hides the wait window and "unlocks" the GUI.

5.4.3 Layouting the Graph

Layouting the extensional navigation graph is managed by a special object belonging to the `LayoutTouch` class. The timer calls repeatedly (about 25 times per second) a special method (`Run()`) of the class, in the background of the main thread, which assures smooth animation of the vertices and edges. The implemented algorithm is based on the one introduced in the *LayoutGraph* example shipped with the Java JDK and *TouchGraph*.

Roughly speaking the process of layouting the graph is performed in the following steps, executed by the following methods (started from the `Run()` method of the `LayoutTouch` object):

- `void relaxEdges()` – calculates nodes' motion vectors to pull them close together,

- `void avoidLabels()` – move nodes in such a way to avoid labels,
- `void moveNodes()` – moves nodes according to the predefined motions' vectors.

5.4.4 Working with Object

An object content is presented using instance of the `ObjectViewerForm` class. Similarly to the starting extensional navigation, user starts the process by executing appropriate command from the context menu. As the result the `ShowObject(AdUniqueID objectID)` method from the `MainFrame` class is run. The method creates a new thread, which asynchronously presents a selected object. Figure 88 illustrates the process:

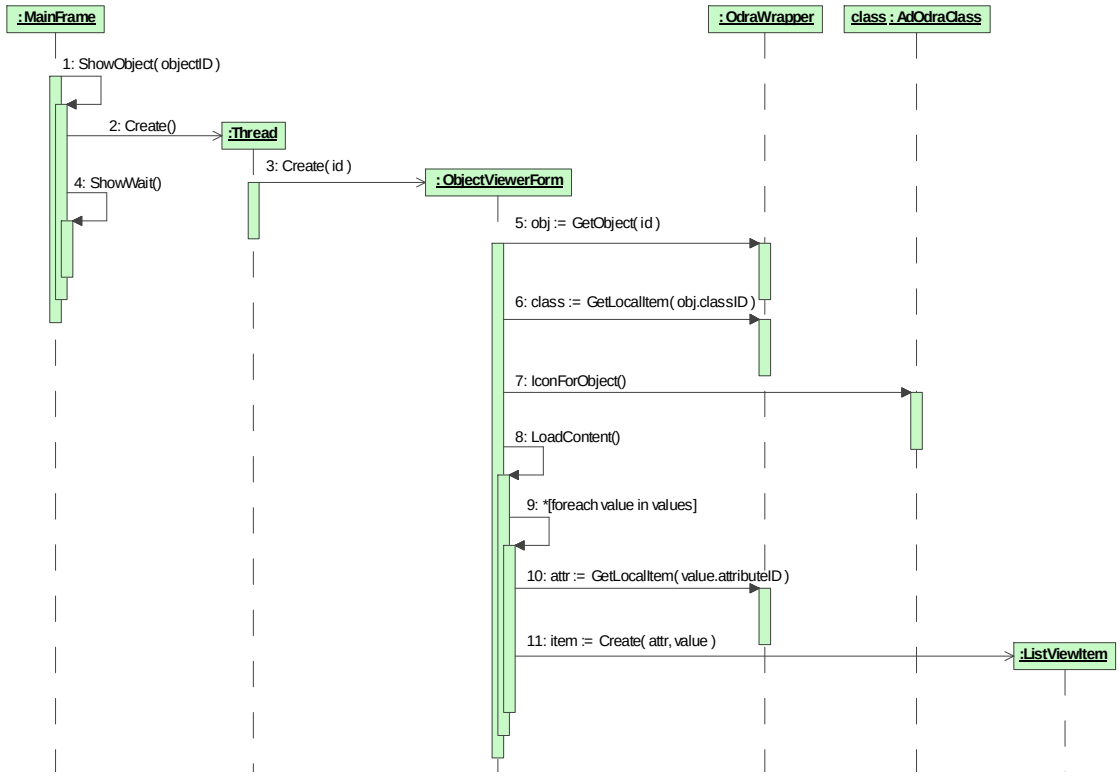

Figure 88. Sequence diagram for showing object.

- Call 3 creates a new instance of the `ObjectViewerForm` class,
- Call 5 retrieves an object from the wrapper, and call 6 its class,
- Call 7 gets icon for the object (which will be shown to the user),

- Call 8 load content of the object, which means values (as strings) of all its attributes,
    - For each value (call 9), its attribute is retrieved (call 10)
    - And new instance of the `ListViewItem` class is created (storing attribute and its value).

5.5 Baskets

Baskets are shown using an instance of the `BasketForm` class. However they are stored inside tags (`Tag` property) of the `TreeViewMS` object (this is a modified version of the core C# `TreeView` control). This object is visualized on the Figure 47 (page 70). Aside of the mentioned classes, basket implementation is based on the following classes:

- `BasketItem` – main super class for basket items. Assures basic functionality like storing referencing tree node, label, associated tool tip (declaration only) and overridden version of the `ToString()` method,
- `BasketFolder` – stores information about sub/super basket. Mainly implements overridden versions of the tool tip and description properties,
- `BasketObject` – records the information about a single object in the basket. Contains an overridden version of the tool tip and comment properties, and also id of the object. That is, baskets store only object ids.

As mentioned earlier, baskets also support drag and drop technique. However, we are not going to describe it, because the implementation is based on rules already discussed in chapter 5.3.5.

5.5.1 Creating Basket

Creating of a new basket is started by clicking the button *Create* in the window shown on Figure 47 (page 70). The activity starts `TreeNode CreateBasket (TreeNode node)` method of the `BasketForm` class, which acts as follow (Figure 89):

- Call 2 create a new instance of window, which allows entering name and description of the basket,
- Call 3 shows above window,
- Call 4 creates tree node and call 5 object, which stores information about new basket,
- Call 6 and 7 ensures visibility of the new basket in the tree,
- Call 8 returns the newly created node (containing `BasketFolder`).

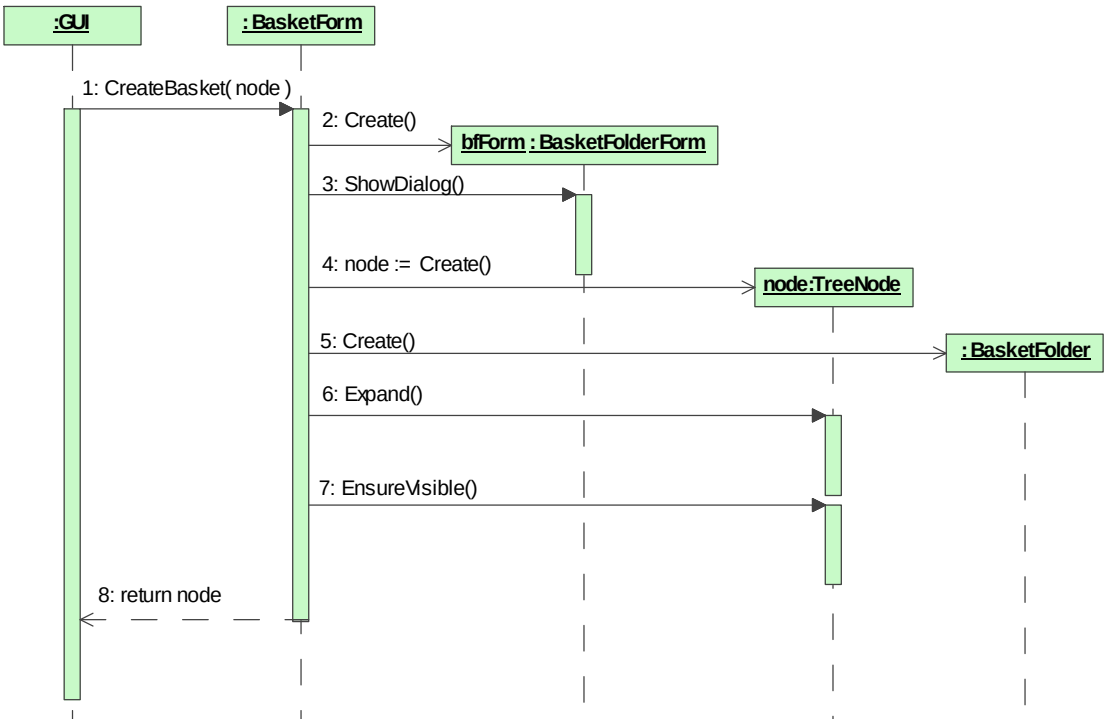

Figure 89. Sequence diagram for creating a new basket.

5.5.2 Basket's Operations

Performing basket's operations is possible from the context menu. Choosing a suitable command starts an appropriate method (the names are self explained):

- `void PerformSumOfBasket(...);`
- `void PerformIntersectionOfBasket(...);`
- `void PerformDifferenceOfBasket(...);`

All of them get three parameters:

- two of them contain instances of the `TreeNode` class, which are the arguments of the operations,
- the third one is also a `TreeNode` instance, which will have the result of the operation.

The methods work in the similar way, so we will discuss their implementation using only one of them: `void PerformIntersectionOfBasket(TreeNode nodeA, TreeNode nodeB, TreeNode nodeResult)`. Figure 90 contains sequence diagram illustrating performed actions:

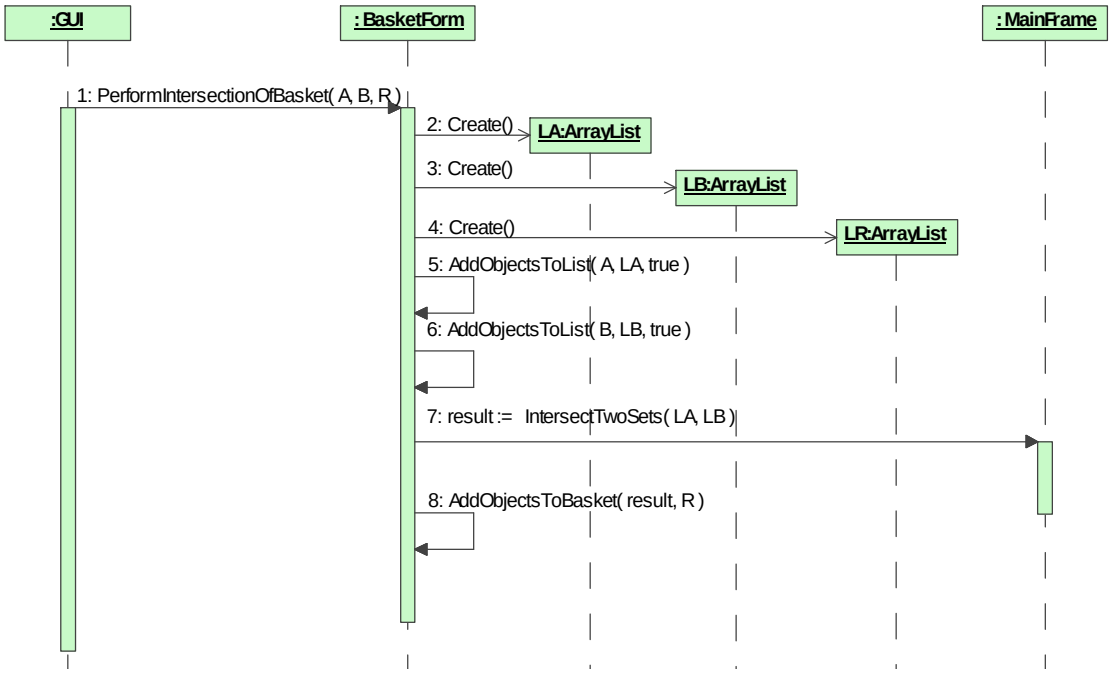

Figure 90. Sequence diagram illustrating operation on baskets.

- Calls 2 and 3 create `ArrayLists` to store operation's parameters.
- Call 4 creates another `ArrayList` for the results.
- Calls 5 and 6 add basket's items (from all nested baskets) as objects to lists. Thus, each of lists contains objects' ids from all nested baskets (without information about sub-baskets). The third method's parameter indicates if repetitions are disallowed.

- Call 7 runs method, which performs appropriate operation (intersection) on two lists. The result is returned.
- Call 8 converts the result containing objects' ids to the appropriate tree constructs (instances of the `TreeNode` linked with `BasketObjects`).

5.5.3 Baskets' Persistency

Basket's persistency is implemented using the C# native serialization. Unfortunately, the `TreeView` control does not support serialization itself. Thus, manual methods have been introduced:

- `void SaveBasket(Stream stream);` saves basket's content to the given stream. Calls class method (static) `void SaveTree(TreeView tree, Stream stream)`.
- `void LoadBasket(Stream stream);` loads basket's content from the given stream. Calls class method (static) `void LoadTree(TreeView tree, Stream stream)`.

Roughly speaking above methods utilize instances of the `BinaryFormatter` class to load/save items, which the contents consist of. Each of the items is read/written in turn, one by one from/ to the stream.

5.6 Active Extensions

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 Mavigator. When we started thinking about the way of implementation, two different solutions came to our minds:

- Our first idea was to define some programming interface, which should be implemented by a particular class created by the programmer,
- The second approach has been based on implementing only one method with special parameters.

Finally we have found that the first approach would be exaggeration. Thus we have decided to follow the second one. Programmer who wants to develop a particular

Active Extension has to create only one method (in special class): `public`, `static`, with two parameters:

- instance of the data wrapper,
- collection containing ids 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, system adds this method to the list of created extensions. When the user wish to run a particular Active Extension, the system starts an associated method, passing instance of the data wrapper and collection of objects' ids'.

The whole process of managing Active Extensions is performed by the instance of the `ActiveExtensionEngine` class. Its main methods are enumerated below:

- `bool Compiled;` Indicates if an assembly is compiled and ready to run,
- `String SourceCode;` Gets or sets a source code as a C# string,
- `void AddAssemblyReference(String referenceName);` Adds an assembly reference (needed by classes utilized in Active Eextensions' methods,
- `void Run(String methodName, IList IDs, AdWrapper wrapper);` Runs a method with the given name. Passes two parameters to the method: list of `uniqueID`s and instance of the wrapper,
- `bool Compile();` compiles the source and creates assembly (ready to run) in memory. Utilizes instances of the following classes (or interfaces):
  - `CSharpCodeProvider`,
  - `ICodeCompiler`,
  - `CompilerParameters`,
  - `CompilerResults`,
  - `Assembly`.

- `IList MethodsNames;` Gets a list containing names of all methods, which have two parameters: `IList`, `AdWrapper`. Each of them is treated as a single Active Extension. Thus Active Extensions names must be unique,
- Couple of methods to handle error messages.

The AE engine also provides some helper methods (grouped in the `ActiveExtensionsUsefulMethods` class, as `public` and `static`), which could be useful in implementing own extensions:

- `double GetMinValue(AdWrapper wrapper, IList IDs, AdAttribute attribute, ref AdObject minObject, ref int counter);` Returns the minimum value of a given attribute for a given set of objects,
- `double GetMaxValue(AdWrapper wrapper, IList IDs, AdAttribute attribute, ref AdObject maxObject, ref int counter);` Returns the maximum value of a given attribute for a given set of objects,
- `String GetAttributeValue(AdWrapper wrapper, AdUniqueID objectID, AdAttribute attribute);` Returns a value (as a `String`) of a given attribute for a given object (by its ID),
- `String GetAttributeValue(AdWrapper wrapper, AdObject obj, AdAttribute attribute);` Returns a value (as a `String`) of a given attribute for a given object (by its instance),
- `bool IsClassContainsAttribute(AdClass theClass, AdAttribute atr);` Indicates if a given class contains a given attribute.

Roughly speaking, a programmer, who would like to create an own Active Extension, has to perform only few simple steps:

- define a new method (`public`, `static`) with two parameters: `IList IDs, AdWrapper wrapper` and put it into `ActiveExtensionInstance` class. The name of the method could by

any kind, upon the condition that will be unique among other methods in the class,

- fill the method's body with an appropriate source code,
- add appropriate using statements before the class definition.

```
public static void ShowNumberOfMarkedObjects(IList IDs, AdWrapper wrapper)
{
    if(IDs == null)
    {
        MessageBox.Show("There are no marked objects defined for this class!",
                                    "Wrapper version " + wrapper.Version);
    }
    else
    {
        MessageBox.Show("Number of marked objects: " + IDs.Count,
                                    "Wrapper version " + wrapper.Version);
    }
}
```

Example 18. Source code of the simple Active Extension.

Example 18 presents the source code of the very simple Active Extension, which shows the number of objects passed for processing. Two things are worth noting:

- the number of objects is retrieved from the collection's property: `IList::Count`,
- showing a wrapper version illustrates wrapper utilization.

6 Mavigator's Evaluation

According to [Shn03] and [Nor04], the application's evaluation process has four main aims:

- Discovering major problems that could result in a human error or lead to frustration of the user,
- Reducing training time,
- Increasing performance and efficiency,
- Improving user's satisfaction from using the software.

Application's usability could be defined as an ability to satisfy user's needs related to the application. As noted in [Mur00b] there are two fundamental approaches to usability:

- By principles. This means choosing such usability principles, which will be adequate for particular kind of application and user. Some of them have been described in Chapter 4.7 (page 83),
- By evaluation. This approach requires evaluation by the users, which means that the whole application (or at least some part of them) must be developed. The easy part of the method is criticizing current solutions. Unfortunately the hard one is deciding what to change to improve it.

We believe that applications' developers have to try joining two above approaches. During the entire application's creation process, all known usability principles must be taken into consideration. Still, after developing a "beta" version, evaluation with users must be conducted. During researches for this thesis we follow this rule.

In [Pla04] Plaisant distinguish four thematic areas of evaluation:

- Controlled experiments comparing design elements. The studies in this category might compare specific widgets or compare mappings of information to graphical display,

- Usability evaluation of a tool. Those studies might provide feedback on the problems that users encountered with a tool and show how designers went on to refine the design,
- Controlled experiments comparing two or more tools. Those studies usually try to compare a novel technique with the state of the art.
- Case studies of tools in realistic settings. This is the least common type of studies. The advantage of case studies is that they report on users in their natural environment doing real tasks, demonstrating feasibility and in-context usefulness. The disadvantage is that they are time consuming to conduct, and results may not be replicable.

Our first prototype SKGN has been used in the European project ICONS[1]. Thus it has been informally evaluated during using for the Structural Fund Projects Portal. Hence we have some informal response from the users, generally very positive. However, this thesis requires more formal evaluation.

We have decided to conduct usability evaluation of the tool (the second area of evaluation). Next sub-chapters give detailed description of the process and discuss the results.

6.1 Procedure

Mavigator's evaluation has been conducted by the 6 subjects in three groups (because of problems with scheduling). All of them were students from the Polish-Japanese Institute of Information Technology (different departments). All surveys were in Polish and based on [Hol93], [Nor00].

At the beginning, subjects have to answer questions about their experience and knowledge. All answers were from range 0 (lack of knowledge) to 10 (expert). Below we have enumerated all of them (question 1.1 is just an id of the subject):

1.2 Object-oriented database concepts (classes, associations, etc.),
1.3 Microsoft Access experience,
1.4 Textual query language experiences (i.e. SQL, OQL),

[1] EU 5th Framework project ICONS (Intelligent Content Management System), IST-2001-32429

1.5 Visual information retrieval system experience,

1.6 Programming language experience.

Then, all subjects were trained on the Mavigator prototype. The training program consisted of:

- Short introduction with description of motivations behind Mavigator,
- Review of background concepts like:
  - database schema graph,
  - classes, associations, attributes, etc.
- Discussion of Mavigator's key concepts:
  - Intensional navigation,
  - Extensional navigation,
  - Baskets,
  - Active Extensions,
  - Virtual Schemas.
- Detailed instructions on using the prototype. This includes demonstration of using particular techniques like filtering, navigating, etc.

After the training, subjects have to find answers for particular query questions. As query questions, the case studies described in Chapters 4.8.1 - 4.8.8 (pages 84 - 90) have been utilized. For each query question, each subject has to fulfil survey with the following items (2.1 is an id of the subject, 2.2 is the number of the query question):

2.3 Difficulty of the query question – from 1 (easy) to 10 (very hard),

2.4 Success - in percents,

2.5 Time to completion – in minutes,

2.6 Remarks.

At the end of evaluating, after solving all query problems, each user answers summary questions about the metaphors, prototype, etc:

3.1 Comprehensibility – from 1 (confusing) to 10 (clear),

3.2 Ease of use – from 1 (difficult) to 10 (easy),

3.3 Speed of use – from 1 (slow) to 10 (fast),

3.4 Performance – from 1 (slow) to 10 (fast),

3.5 Overall satisfaction – from 1 (terrible) to 10 (wonderful),

3.6 Remarks,

3.7 Ideas for improvement,

3.8 Ideas for new Active Extensions.

6.2 Results

Detailed results could be found in appendix D on page 160. Next sub-chapters contain summary of the results.

6.2.1 Subjects

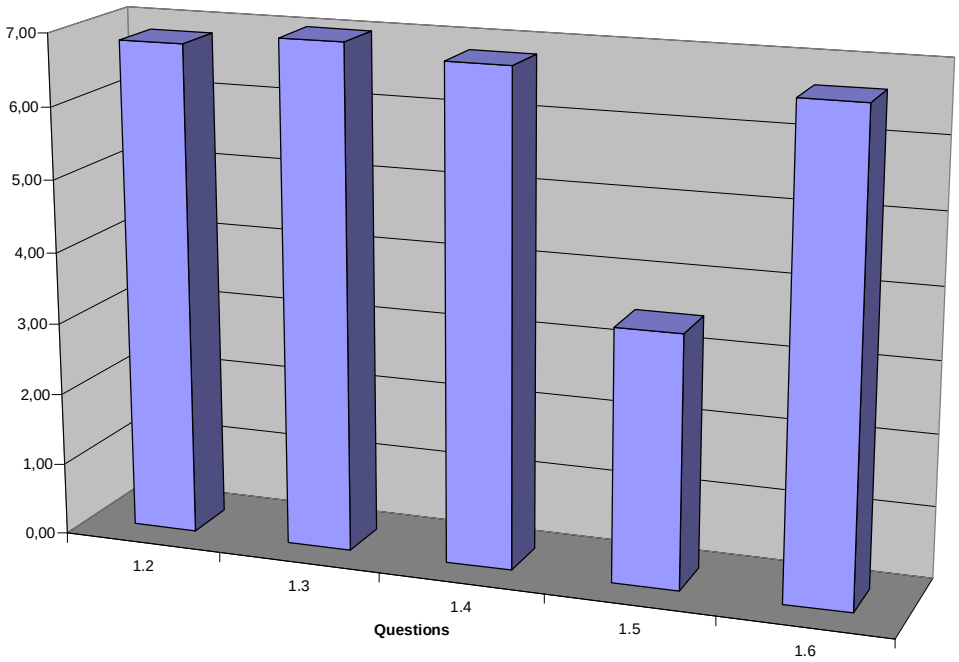

Figure 91. Average information about subjects.

Figure 91 shows average values given by subjects describing themselves. Bars refer to the questions shown on page 138. According to the answers, all subjects have been quite familiar with database concepts (question 1.2, page 138), MS Access (question 1.3, page 138), textual query languages (question 1.4, page 138) and programming language (question 1.6, page 139). However, most of them were not familiar with graphical user interfaces (question 1.5, page 139). From the testing point

of view, maybe it would be better to find subjects less familiar with these topics. However, we have to take into account that some of the answers could be a little bit exaggerated. This might be caused by the fact that subjects were students (and the administrator of the experiment was their teacher), who should be familiar with the mentioned terms.

6.2.2 Query Questions

Figure 92 presents chart showing two kinds of information (for each query question):

- Average difficulty of the query questions (bars),
- Average time to completion (line).

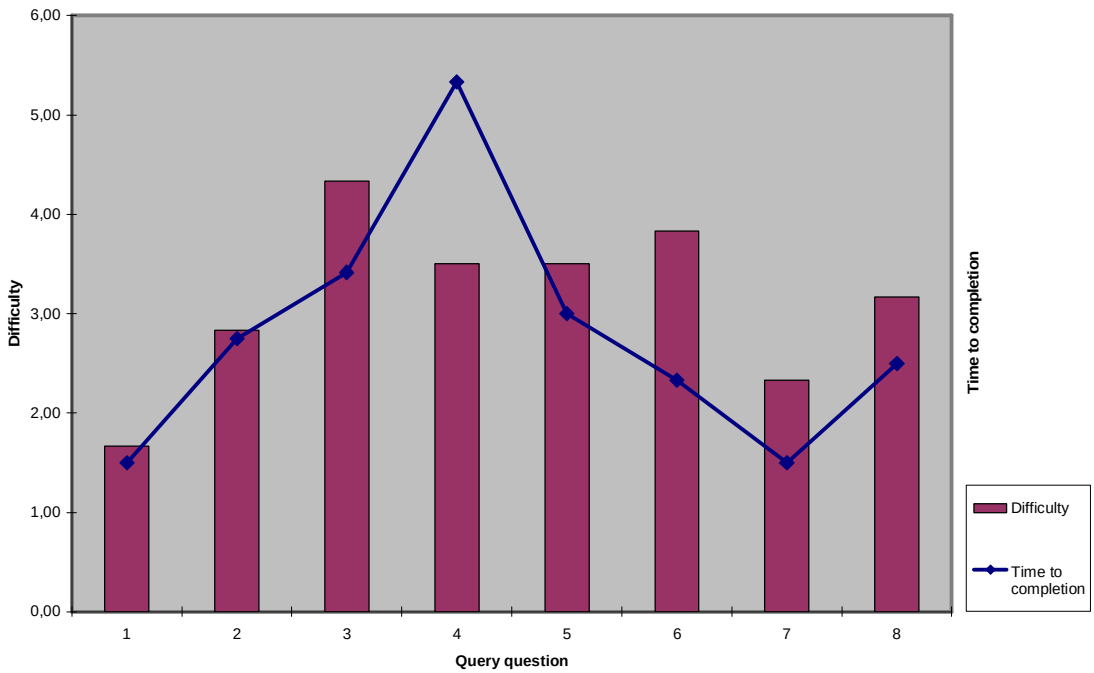

Figure 92. Chart showing dependence between questions' difficulty and time for completion.

As it can be seen, queries, which has been judged as harder, take more time to complete (except the fourth one). Also, almost all queries (without one – for one subject), have been completed in 100 percent (not shown on the chart). The shortest time to complete was 1.5 min. (for number 1 and 7), the longest one was 5.5 min. (for number 4) and an average time to complete was about 2 minutes and 45 seconds.

According to the subjects' answers, the most harder question was number 3 and then number 6. However, even the hardest ones, still have been judged as easier than medium (less then 5 in the 10 degrees scale).

6.2.3 Overall Results

Figure 93 shows average answers (to the questions from page 139), given by the subjects, about Mavigator's prototype. Roughly speaking, all answers are positive (more then 5 in the tenth degrees scale).

Comprehensibility (question 3.1), which is a very important factor in graphical user interfaces, has average value 6.5 with minimum at 4 (one subject) and maximum at 8 (also one subject). Most subjects give 7/10.

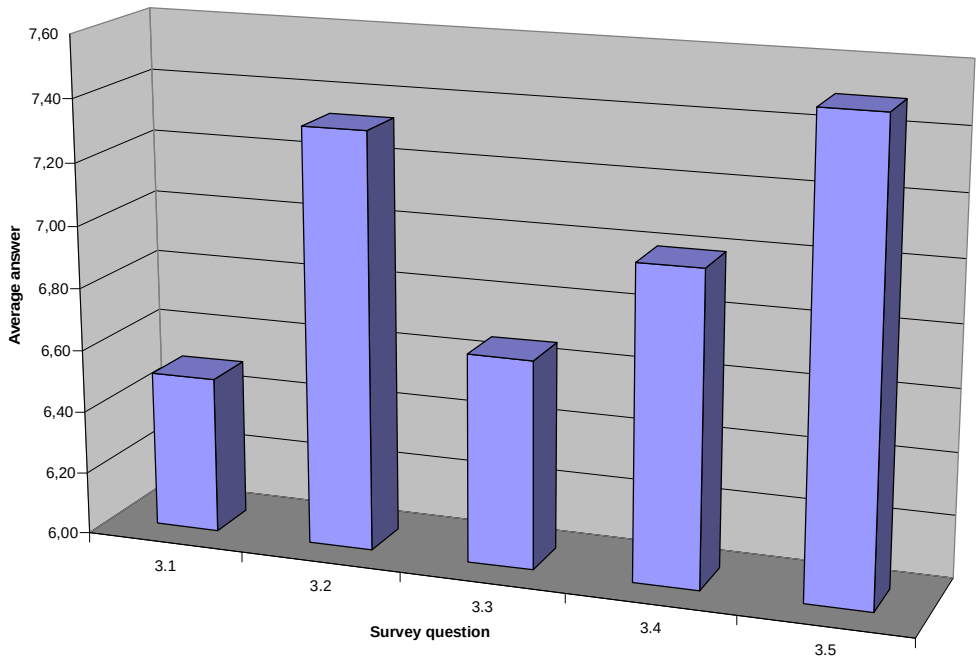

Figure 93. Chart showing average answers about Mavigator.

Another important factor, which has been judged by the subjects, is ease of use (question 3.2). In this case, marks vary from 6, through 7 and 8 up to the 9. Average value was 7.3/10, which is quite high.

Speed of use (question 3.3), which is different from speed of working (performance – 3.4), describes things like the way of using options, running actions, etc.

This factor is related in some way to the comprehensibility, and has achieved average value about 6.6/10.

Speed of working (question 3.4) is not so important in case of prototypes. Moreover performance of the information retrieval tools (including Mavigator) is tightly connected with the performance of the data source. The current prototype works with the ODRA experimental database server, which has not been optimized yet. However, in this field Mavigator also got quite a good result: 7/10.

And the last factor, which has been assessed - overall satisfaction of the user. This is very subjective judgment and has big impact on the user's decision: do I really want to use the tool? Fortunately, Mavigator's users give them average mark about 7.5/10 with minimum at 7 and maximum at 9. Such a result bears good testimony to the ideas and metaphors behind Mavigator.

6.2.4 Users' Remarks

Subjects have the right (and they have been encouraged to) to makes comments and remarks. All of them have been carefully analyzed. Most of them were related to the graphical user interface (and have been taken into account) and not to the metaphors itself. Next subchapters discuss the most important ones.

6.2.4.1 Items' Lists

One subject noticed that a list of attributes during defining a single predicate is not sorted any way. As a result all lists (of items of any kind) appearing in the prototype have been analyzed and sorted alphabetically.

6.2.4.2 Showing Objects

Another subject gave attention that one of the most frequently performed operations is showing marked objects. Thus, it would be nice to have an opportunity to perform it very quick. Hence, a new way of showing marked objects has been added: when a user double clicks on the class icon, a list of marked object is shown.

6.2.4.3 Showing Basket

Next problem was connected with showing basket's window. When the window has been shown, and then another window covered them, choosing *Show basket* from

the menu, has no effect. It has been fixed by bringing the basket's window to the front (after selecting appropriate option form the menu).

6.2.4.4 Working with Single Predicate

When a user defines a single predicate (for filtering some objects; see Figure 32 on page 57), there is a necessity to enter some values. One of the subjects suggests that application should prompt values of the selected attributes read from existing objects. The idea is obviously good. However, its implementation could lead to serious performance problems. This is caused by the fact that prompting values requires reading all objects from the database (or at least all values of the particular attribute). This is technically possible, but the performance overhead makes it questionable.

6.2.4.5 Help System

A few subjects ask for a help system. Of course this is a must in a commercial application. However, due to the prototype nature of the project, such a system has not been implemented.

6.2.4.6 Marking objects with filtering

During filtering objects, one of the subjects, reported that filtering system does not work. After the short investigation, come out that the user is filtering from 0 marked objects, which leads to marking 0 objects. As a result, a dedicated message has been added, which informs the user about the situation.

6.3 Summary

Conducted surveys satisfied two important evaluation's targets:

- Allowed improving the prototype,
- Proved rightness of the Mavigator's ideas and metaphors.

Overall high marks, issued by the prototype's users confirm Mavigator's high usability and easy-in-use.

7 Conclusion and Future Work

The goal of this thesis was to make an investigations into visual metaphors, which allow naïve users (computer non-professionals) working with an object-oriented database. The research had shown that such metaphors must fulfilment following requirements:

- Expressive power. The user must be able to perform queries similar to ones expressed in textual languages.
- Easiness of use. Because of special kind of the user, the proposed metaphors must be easy in use and intuitive.
- Information filtration. Nowadays domains are very complicated. Thus database schema graphs, describing the domains are complicated too. Hence there must be a way to limit shown graph to parts, which are really interested to the user.
- Customization. There must be a way to customize a database schema graph, including classes' and attributes' names, adding/hiding associations, etc.
- Flexibility. It is quite obvious that it is not possible to create (or even define) functionality, which suits needs of all users. So, there must be a way to add new functions to the existing ones. In particularly, those modifications should allow changing the way of presenting query results.
- User awareness. During the whole process of information retrieval, the user must be aware of all performed actions.
- Coherence. All the utilized metaphors must be able to exchange information. The same entities existing in different metaphors must be treated exactly in the same way.
- Interoperability. Information is retrieved by some purposes. Sometimes, it is enough to consume the information inside the metaphors, but there must be also a way to utilize information in external systems.
- Result's recording. A user must be able to store any kind of information (not only final results), which has been retrieved during the process.

7.1 Our Proposal

Our proposal, called Mavigator, fulfils all the previously enumerated requirements. The implemented prototype proves that each of them is reflected in one or more Mavigator's metaphors:

- Intensional navigation allows navigation in a database schema graph, where classes act as vertices and associations as edges. The user can mark objects, which are the result of the query. Marked objects could be viewed, stored or subsequently processed.
- Extensional navigation enables navigation between particular objects. It is possible to move from one object to another one connected by a link (an instance of an association).
- Baskets make it possible to store objects. Each basket could contain other (nested) baskets or objects. The main basket as well as its sub-baskets is persistent and assigned to the particular user.
- The Virtual Schemas module allows redefining a database schema graph (they acts as some kind of database views, but on the application side rather on the database server side). Changes may include renaming classes or attributes, adding or removing associations, and others.
- Active Extensions, implemented by the programmer, make it possible to add new functionalities to the application, even without stopping it. Particular utilizations include adding new functions operating on database objects (average, minimum, maximum)[2], new ways of analyzing data (Active Projections)[2] or exporting data (Objects exporter)[2].

In our prototype all the above metaphors are supported by user-friendly GUI widgets like tool tips, combo boxes, etc. Also we have introduced a dedicated solution to visually formulate logical conditions for filtering objects.

Moreover, the presented metaphors treat same items in exactly the same way: for instance, an object utilized in a basket has the same set of common options like in extensional navigation or in the specialization of Active Extension called Active

[2] Items in brackets have been implemented in the prototype.

Projections. Such an approach improves not only user's awareness but also the explicit power of the entire solution.

7.2 Future Work

The Mavigator prototype shows basic visual retrieval metaphors that perhaps in a similar manner can be implemented in other systems addressing structural data. Mavigator is fully functional tool. However, due to the nature of this work, some changes and improvements could be made. Future works could concerns following topics:

- Adding new functions to the core metaphors, for instance adding querying mode to the intensional navigation. In the present solution, when a user has to navigate from the *A* class to *B* and then from *B* to *C*, to find objects from the *C* class, all intermediatory objects are downloaded and marked. It could be useful to *record* user's actions and then to execute them in one click without downloading intermediatory results,
- New wrappers to other data sources. The current prototype works with the ODRA database management system. Some other wrappers to other database systems, e.g. to relational or XML-oriented, could be implemented,
- Performance improvements. As mentioned previously, the entire prototype has been developed to keep clarity of the solutions (and code) rather then performance. In real life application it should be changed,
- Moving Virtual Schema module above the wrapper level. Such an approach makes the module independent from particular data source and a dedicated wrapper.

8 Bibliography

[Ahl92] Ahlberg, C., Williamson, C., Shneiderman, B.: Dynamic queries for information exploration: An implementation and evaluation. In Proceedings of the Conference on Human Factors in Computing Systems (SIGCHI '92), pp. 619-626. ACM Press. 1992

[Bar03] Barclay, P.J., Griffiths, T., McKirdy, J., Kennedy, J., Cooper, R., Paton, N.W. and Gray, P.: Teallach - A Flexible User-Interface Development Environment for Object Database Applications, J. Visual Languages and Computing, 14(1), 47-77, 2003.

[Bat91] Batini C., Catarci T., Costabile M.F., Levialdi S.: Visual Strategies for Querying Databases. Proc. of the IEEE Int. Workshop on Visual Languages, Japan, October 1991.

[Ber83] Bertin J.: Semiology of Graphics. Madison, Wis.: Univ. of Wisconsin Press, 1983. Translated by W. J. Berg

[Car96] Carey M.J., Haas L.M., Maganty V., Williams J.H.: PESTO: An Integrated Query/Browser for Object Databases. Proc. VLDB (1996) 203-214

[Cas01] Cassel K., Risch T.: An Object-Oriented Multi-Mediator Browser. 2nd International Workshop on User Interfaces to Data Intensive Systems, Zürich, Switzerland, May 31 - June 1, 2001

[Cat00] Catarci T.: What Happened When Database Researchers Met Usability. Information Systems 25(3), 2000, 177-212.

[Chi92] Chimera, R.: Value Bars: An Information Visualization and Navigation Tool for Multi-Attribute Listings. Proc. ACM CHI ‘92, pp. 293-294, (1992).

[Coo00] Cooper, R., McKirdy, J., Griffiths, T., Barclay, P., Paton, N.,

Gray, P., Kennedy, J. and Goble, C.: Conceptual Modelling for Database User Interfaces. In Arisawa, H. & Catarci, T. (Eds), Visual Database Systems - VDB5, 2000 pp129-138. : Kluwer Academic Publishers.

[Der97] Derthick M., Kolojejchick J., Roth S.F.: An Interactive Visual Query Environment for Exploring Data, Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '97), ACM Press, (1997) 189-198

[Fer99] Fegaras L.: VOODOO: A Visual Object-Oriented Database Language For ODMG OQL. ECOOP Workshop on Object-Oriented Databases 1999, 61-72

[Gri01] Griffiths, T., Barclay, P., Paton, N., McKirdy, J., Kennedy, J., Gray, P., Cooper, R., Goble, C., da Silva P.: Teallach: Model Based User Interface Development Environment for Object Databases. Interacting with Computers 14 (2001), pp 31-68, Elsevier.

[Hol93] Holyer A.: Methods for Evaluating User Interfaces. Cognitive Research Paper No. 301, School of Cognitive and Computing Sciences, University of Sussex, Brighton, 1993.

[Jon99] Jones S., McInnes S.: A graphical user interface for boolean query specification. International Journal on Digital Libraries Special Issue on User Interfaces for Digital Libraries, 2(2/3):207–223, 1999

[Jos02] Josifovski V., Risch T.: Query Decomposition for a Distributed Object-Oriented Mediator System. Distributed and Parallel Databases J., 11(3), pp 307-336, Kluwer, May 2002.

[Koz03] H.Kozankiewicz, J.Leszczyłowski, K.Subieta. Updateable XML Views. Proc. of ADBIS'03, Springer LNCS 2798, 2003, 385-399

[Kum97] Kumar, H., Plaisant, C., Shneiderman, B.: Browsing

Hierarchical Data with Multi-Level Dynamic Queries and Pruning. IJHCI, vol. 46, pp. 103-124, (1997).

[Mit96a] Mitchell K., Kennedy J., Barclay P.: A framework for user-interfaces to databases, in Procs of Workshop on Advanced Visual Interfaces, ACM press (1996)

[Mit96b] Mitchell K., Kennedy J.: DRIVE: An environment for the organised construction of user interfaces to databases, 3rd International Workshop on Interfaces to Databases, Springer-Verlag Electronic WIC (1996).

[Mur00] Murray N., Paton N.W., Goble C.A., Bryce J.: Kaleidoquery - A Flow-based Visual Language and its Evaluation. Journal of Visual Languages and Computing 11(2), 2000, 151-189

[Mur00b] Murphu N.: Principles of User Interface Design. Internet Appliance Design Article, December 2000, http://www.embedded.com/2000/0012/0012ia1.htm.

[Mur98] Murray N., Goble C., Paton N.: Kaleidoscape: A 3D Environment for Querying ODMG Compliant Databases. In Proceedings of Visual Databases 4, L'Aquila, Italy, May 27-29, 1998

[Nor00] North C. L., A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization. PhD Dissertation, Graduate School of the University of Maryland, College Park, 2000.

[Nor04] Norman K. L., Panizzi E.: Levels of Automation and User Participation in Usability Testing, University of Maryland, Laboratory for Automation Psychology and Decision Processes, Technical Report: LAP-2004-01, HCIL-2004-17, 2004.

[Now98] Nowell L.T.: Graphical Encoding for Information Visualization: Using Icon Color, Shape, and Size To Convey

Nominal and Quantitative Data. PhD dissertation, Virginia Polytechnic Institute and State University, 1998

[ODM00] Object Data Management Group: The Object Database Standard ODMG, Release 3.0. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000

[Pla04] Plaisant C.: The Challenge of Information Visualization Evaluation. In Proc. of Conf. on Advanced Visual Interfaces AVI'04 (2004)

[Rot94] Roth, S. F., Kolojejchick, J., Mattis, J., Goldstein, J.: Interactive graphic design using automatic presentation knowledge. In Proceedings of the Conference on Human Factors in Computing Systems (SIGCHI '94), pp. 112-117. 1994

[Rot97] Roth F., Chuah M., Kerpedjiev S., Kolojejchick J., Lucas P.: Towards an Information Visualization Workspace: Combining Multiple Means of Expression. Human-Computer Interaction Journal, Volume 12, Numbers 1 & 2, 1997, 131-185.

[Shi81] Shipman D.: The functional data model and the data language DAPLEX. ACM Transactions on Database Systems, vol. 6, no. 1, 1981.

[Shn03] Shneiderman B., Plaisant C.: Designing the user interface: Strategies for effective human-computer interaction (4th Ed). Reading, MA: Addison-Wesley, 2003.

[Shn91] Shneiderman, B.: Visual user interfaces for information exploration. In Proceedings of the 54th Annual Meeting of the American Society for Information Science, pages379–384, Medford.NJ, 1991.Learned Information Inc.

[Smi02] Smith M., King P.: The Exploratory Construction Of Database Views. Research Report: BBKCS-02-02, School of Computer Science and Information Systems, Birkbeck College,

University of London, 2002

[Sto02] Stolte Ch., Tang D., Hanrahan P.: Polaris: A System for Query, Analysis and Visualization of Multidimensional Relational Databases. IEEE Transactions on Visualization and Computer Graphics, Vol 8, No 1, January-March 2002

[Sub04] Subieta K.: Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, Warsaw 2004, ISBN 83-89244-29-2, 522 pages (in Polish)

[Sub95] Subieta K., Beeri C., Matthes F., Schmidt J.W.: A Stack-Based Approach to Query Languages. Proc. 2nd East-West Database Workshop, 1994, Springer Workshops in Computing, 1995, 159-180

[Trz04a] Trzaska M., Subieta K.: Structural Knowledge Graph Navigator For The Icons Prototype. Proc. of the IASTED International Conference on Databases and Applications (DBA 2004)

[Trz04b] Trzaska, M., K.Subieta, K.: The User as Navigator. Proc. 8th East-European Conference on Advances in Databases and Information Systems (ADBIS), September 2004, Budapest, Hungary

[Trz04c] M. Trzaska, K. Subieta: Usability of Visual Information Retrieval Metaphors for Object-Oriented Databases. Proceedings of the On The Move Federated Conferences and Workshops (DOA, ODBASE, CoopIS, PhD Symposium), Springer Lecture Notes in Computer Science (LNCS 3292), pp. 822-833, October 25-29, 2004, Larnaca, Cyprus.

[Wal] Walrus - Graph Visualization Tool, http://www.caida.org/tools/visualization/ walrus/

[Zlo77] Zloof M. M.: Query-by-Example: A Database Language. IBM

Syst. Journal, 16(4), 1977, 324-343

9 Appendices

A - Abbreviations
B - List of figures
C - List of examples
D - User Study Materials
E - Extended Abstract (in Polish)

A. Abbreviations

DBMS	Database Management System
GUI	Graphical User Interface
MDI	Multi Document Interface
OCL	UML's object constraint language
OQL	Object Query Language
SDI	Single Document Interface
UML	Unified Modeling Language
UI	User Interface
XML	eXtensible Markup Language

B. List of figures

C. List of examples

D. User Study Materials

1. Information about subjects

1.1 Subject id

1.2 Object-oriented database concepts (classes, associations, etc.) – from 0 (lack of knowledge) to 10 (expert).

1.3 Microsoft Access experience – from 0 (lack of knowledge) to 10 (expert).

1.4 Textual query language experiences (i.e. SQL, OQL) – from 0 (lack of knowledge) to 10 (expert).

1.5 Visual information retrieval system experience – from 0 (lack of knowledge) to 10 (expert).

1.6 Programming language experience – from 0 (lack of knowledge) to 10 (expert).

1.1	1.2	1.3	1.4	1.5	1.6
1	7	8	7	1	7
2	7	7	7	2	8
3	**6**	**5**	**5**	**6**	**5**
4	**7**	**9**	**8**	**7**	**9**
5	**6**	**5**	**5**	**5**	**5**
6	**8**	**8**	**9**	**0**	**6**

2. Information about query questions

2.1 Id of the subject

2.2 Number of the query question (shown as columns in the above tables)

2.3 Difficulty of the query question – from 1 (easy) to 10 (very hard)

2.4 Success - in percents

2.5 Time to completion – in minutes

2.6 Remarks (not shown because they were in Polish. However the most important ones have been discussed in chapter 6.2.4 on page 143)

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**1**							
2.3	1	2	3	2	3	2	3	3
2.4	100	100	50	100	100	100	100	100
2.5	0,5	1	5	1	5	2	2	2

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**2**							
2.3	1	2	3	3	5	5	4	4
2.4	100	100	100	100	100	100	100	100
2.5	0,5	0,5	1,5	3	5	3	2	2

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**3**							
2.3	**1**	**2**	**5**	**5**	**3**	**2**	**2**	**3**
2.4	100	100	100	100	100	100	100	100
2.5	**2**	**4**	**5**	**3**	**3**	**2**	**2**	**3**

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**4**							
2.3	**5**	**7**	**8**	**3**	**7**	**6**	**3**	**3**
2.4	100	100	100	100	100	100	100	100
2.5	**1**	**2**	**3**	**1**	**3**	**2**	**1**	**1**

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**5**							
2.3	**1**	**1**	**5**	**5**	**1**	**5**	**1**	**1**
2.4	100	100	100	100	100	100	100	100
2.5	**1**	**1**	**3**	**4**	**1**	**4**	**1**	**2**

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8
2.1	**6**							
2.3	**1**	**3**	**2**	**3**	**2**	**3**	**1**	**5**
2.4	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
2.5	**4**	**8**	**3**	**20**	**1**	**1**	**1**	**5**

3. Overall information

3.1 Comprehensibility – from 1 (confusing) to 10 (clear)

3.2 Ease of use – from 1 (difficult) to 10 (easy)

3.3 Speed of use – from 1 (slow) to 10 (fast)

3.4 Performance – from 1 (slow) to 10 (fast)

3.5 Overall satisfaction – from 1 (terrible) to 10 (wonderful)

3.6 Remarks (not shown because they were in Polish. However the most important ones have been discussed in chapter 6.2.4 on page 143)

3.7 Ideas for improvement (same as above)

3.8 Ideas for new Active Extensions (same as above)

Subject's id	3.1	3.2	3.3	3.4	3.5
1	8	9	8	4	8
2	7	8	7	7	7
3	**7**	**8**	**5**	**7**	**7**
4	**7**	**7**	**7**	**9**	**9**
5	**6**	**6**	**5**	**5**	**7**
6	**4**	**6**	**8**	**10**	**7**

E. Extended Abstract (in Polish)

Użyteczność metafor wizyjnych dla wyszukiwania informacji w obiektowych bazach danych

Celem niniejszej rozprawy było wypracowanie założeń dotyczących użytecznego graficznego interfejsu do wyszukiwania informacji w obiektowych bazach danych. Problemem naukowym wymagającym tu rozwiązania jest pogodzenie prostoty i wygody środków wyszukiwawczych dla powszechnego (tzw. naiwnego) użytkownika z uniwersalnością tych środków. Jak wiadomo, są to sprawy z natury sprzeczne. Badania muszą bardzo poważnie potraktować walor użyteczności praktycznej (*usability*) oferowanego interfejsu do wyszukiwania, jego akceptacji przez użytkownika. Brak tego waloru oznacza nieprzydatność w praktyce, czyli klęskę. Istnieje wiele tego rodzaju klęsk badań w informatyce. Przykładem są różnorodne graficzne języki zapytań, oraz liczne pomysły teoretyczne, takie jak programowanie baz danych przy pomocy reguł logicznych (Datalog), projektowanie baz danych przy użyciu zależności funkcjonalnych i wielowartościowych, itp., które nie zyskały akceptacji użytkowników, mimo że były opracowane i propagowane przez wybitne zespoły naukowe.

Te obserwacje są podstawą metody naukowej, którą zastosowaliśmy w naszej pracy. Polega ona na zbudowaniu prototypu oraz wnioskowaniu na jego podstawie o użyteczności bądź bezużyteczności pewnych pomysłów. Należy podkreślić, że tego rodzaju badania oznaczają działanie w przestrzeni z ogromną ilością arbitralnych wyborów, zaś ocena walorów ich użyteczności jest w dużej mierze subiektywna. Metoda badawcza polega na wypracowaniu pewnej metafory, która nawiązuje do predyspozycji psychologicznych potencjalnego użytkownika, jego naturalnych odruchów i skojarzeń. W przypadku tej rozprawy taką metaforą jest nawigacja, czyli przemieszczanie się pewnego punktu w grafie, mające bezpośrednie skojarzenia z fizycznym poruszaniem się bytów materialnych (ludzi, pojazdów) w pewnej topologii przestrzennej.

Metoda wyszukiwania informacji musi być adekwatna w stosunku do rodzaju danych, które są przeszukiwane, oraz w stosunku do rodzaju docelowego użytkownika tej metody. Obserwacja ta jest szczególnie ważna w przypadku naiwnych

użytkowników (komputerowych nie-profesjonalistów), którzy nie są w stanie (i najczęściej nie chcą) uczyć się wyrafinowanych metod wymagających znacznego wysiłku i znajomości komputerowej egzotyki. W odróżnieniu od silników wyszukiwawczych takich jak Google działających na nie-interpretowanym tekście języka naturalnego, wiele nowych technologii (w szczególności repozytoria XML/RDF lub obiektowe bazy danych) działa na danych strukturalnych, gdzie struktura, nazwy i powiązania danych wyznaczają biznesowe znaczenie danych dla użytkownika. Ponieważ takie technologie są coraz częściej stosowane w biznesie, konieczne są interfejsy użytkownika umożliwiające sprawne i przyjacielskie odpytywanie i przeglądanie takich strukturalnych danych. Jednakże naiwni użytkownicy nie będą w stanie opanować złożonych reguł syntaktycznych, semantycznych i pragmatycznych języków „klawiaturowych", takich jak SQL, OQL lub XQuery oraz specjalnych języków skryptowych dla formatowania rezultatu przeszukiwania. Ten rodzaj użytkownika preferuje przyjacielskie interfejsy graficzne oparte na operacji myszką raczej niż klawiaturą. Obecne oczekiwania powszechnego użytkownika są znacznie zwiększone wskutek pojawienia się wielu bardzo przyjacielskich programów, które są doskonale przystosowane do naturalnych odruchów i skojarzeń użytkowników.

W dysertacji proponujemy zestaw metafor graficznych, które są rezultatem badań nad łatwym w użyciu i jednocześnie bardzo mocnym mechanizmem wyszukiwawczym. Podstawowa tezą dysertacji jest odpowiedź na pięć zasadniczych pytań:

- Jak przedstawić dane dla użytkownika, aby mógł on zrozumieć, co jest przeszukiwane, jaka jest struktura przeszukiwanego zasobu, oraz jak ograniczyć jego pole widzenia tylko do tego fragmentu, które jest mu potrzebny?
- Jakie metafory graficzne są odpowiednie do operacji wyszukiwawczych oraz jak rezultat wyszukiwania ma być zapamiętany?
- Jak rezultat wyszukiwania zaprezentować użytkownikowi?
- Jakich środków należy użyć celem podtrzymania świadomości użytkownika, t.j. takiego jego stanu, w którym nie czuje się on zagubiony, co do celu przeszukiwania oraz etapu, na którym się znajduje?

- Jak rozszerzać funkcjonalność aplikacji, w zależności od potrzeb konkretnego użytkownika?

Te pytania są krytyczne w stosunku do użyteczności całości interfejsu graficznego. Zaniedbanie dowolnego z nich może spowodować niską użyteczność, czyli klęskę danej propozycji mechanizmu wyszukiwawczego.

Odpowiedzi na pytania zostały wypracowane przy pomocy prototypu nazwanego Mavigator. W ramach rozprawy doktorskiej zaproponowaliśmy oraz zaimplementowaliśmy oryginalne rozwiązania, które nie są oczywiście jedyne: stanowią one raczej pewną spójna propozycję nie mającą precedensów w literaturze, opartą o dobrze wyrażoną ideę nawigacji w grafie. Rozwiązania te pozwalają jednak na pewne ogólne wnioski. W zakresie pierwszego pytania proponujemy ograniczenie schematu danych widzianego przez użytkownika przy pomocy wirtualnych perspektyw odnoszących się m.in. do asocjacji pomiędzy danych. Warto zauważyć, że tego rodzaju perspektywy dostarczające wirtualnych asocjacji nie występują jak dotąd w żadnych znanym systemie komercyjnym lub prototypie badawczym (zgodnie z naszą wiedzą). Zaproponowaliśmy moduł nazwany *Virtual Schemas*, w którym perspektyw są definiowane w języku zapytań SBQL. Użycie tego modułu wymaga pewnej wiedzy informatycznej (SBQL jest bardziej interfejsem programistycznym niż językiem dla powszechnego użytkownika), ale zakładamy, że użytkownik końcowy nie musi uczestniczyć w przygotowaniu właściwego mu podschematu - jest to rola administratora bazy danych.

Jako metafory wyszukiwawcze została zaproponowana nawigacja intencyjna (w grafie schematu bazy danych), nawigacja ekstensyjna (w grafie obiektów) oraz trwałe koszyki do przechowywania rezultatów wyszukiwania oraz przeglądania. Metafory te są wzmocnione mechanizmami *drag and drop*, operacjami na zbiorach (suma, przecięcie, różnica zbiorów), klasycznymi warunkami działającymi na atrybutach, oraz innymi opcjami.

Wyprowadzanie rezultatów sesji wyszukiwawczej jest jednym z trudniejszych problemów. Chodzi tu o pogodzenie różnorodnych form wizualizacji danych (często bardzo złożonych, np. wizualizacja wielowymiarowych funkcji) z prostotą interfejsu. Naszym zdaniem, kompromis w tym względzie jest bardzo trudny i prawdopodobnie dla wielu zastosowań w ogóle nie istnieje. Z tego powodu zaproponowaliśmy moduł

Active Extensions, który pozwala na zaprogramowanie w klasycznym języku programowania (aktualnie C#) dowolnej formy wizualizacji (włączając wizualizacje stosowane w eksploracji danych lub animacje) lub funkcji rozszerzającej, a następnie wykorzystanie jej w Mavigatorze. W tym przypadku użytkownik końcowy musi współpracować z profesjonalnym programistą, który przygotowuje dla niego odpowiednie moduły wizualizacyjne. Jest to rozwiązanie rozsądne i być może jedyne możliwe. Pewien zbiór takich modułów wizualizacyjnych oraz funkcji operujących na danych może być przygotowany w momencie tworzenia danej aplikacji i następnie wielokrotnie używany przez jej użytkowników.

Jeżeli chodzi o podtrzymanie świadomości użytkownika, proponujemy pewien zestaw standardowych rozwiązań, takich jak przechowywanie historii akcji użytkownika, możliwość cofnięcia się do poprzednich kroków, hierarchiczna budowa, nazywanie i adnotowanie koszyków rezultatów, itd.

Proponowane rozwiązania są porównane ze stanem sztuki w zakresie interfejsów do wizyjnego wyszukiwania danych oraz wizualizacji danych. Porównanie pokazuje, że proponowane rozwiązania nie mają precedensów w literaturze i pod wieloma względami przewyższają istniejące propozycje. Praca zawiera także rozważania na temat użyteczności generalnie oraz użyteczności proponowanych metafor, poparte dyskusją oraz obserwacjami empirycznymi.

Połączenie wymienionych koncepcji w spójną całość było znacznym wyzwaniem badawczym, zaś wnioski mogą przynieść rozwiązania o dużym znaczeniu dla powszechnej praktyki. Ustawienie wszystkich elementów w spójną i konsekwentną całość stanowiło także swoisty „rebus” techniczny wymagający profesjonalizmu w zakresie technik programistycznych. Przedstawione rozwiązania wymagały także znacznej wiedzy, wyobraźni oraz wielu prób implementacyjnych. Oryginalność i atrakcyjność zaproponowanych rozwiązań została potwierdzona poprzez opublikowanie lub zaakceptowanie czterech artykułów na renomowanych międzynarodowych konferencjach naukowych.