

Chapter 1. MDA and UML

This introductory chapter describes the basic concepts of Model Driven Architecture (MDA) and its base tool - UML.

History

The history of Model Driven Architecture developing totals only some years. OMG consortium (Object Management Group, <http://www.omg.org>) is the main developer of MDA. Today OMG includes more than 800 companies - software and hardware manufacturers. OMG main task is development of standards and specifications regulating application of new information technologies on various hardware and software platforms. OMG takes an active part in development of such popular products and technologies as UML and CORBA.

Now the special attention of OMG consortium is concentrated on MDA technology development.

MDA offers the new integral approach to creation of multiplatform applications. In this chapter we shall review only basic moments necessary for the further understanding. More detailed review of MDA concept can be found in [1].

Structure

MDA is based on three foundations, or elements:

- ❑ UML (Unified Modelling Language);
- ❑ MOF (MetaObject Facility) – the abstract language for description of the models data (metamodels description language);
- ❑ CWM (Common Warehouse Metamodel) – the common standard for the description of information interoperability of data warehouses.

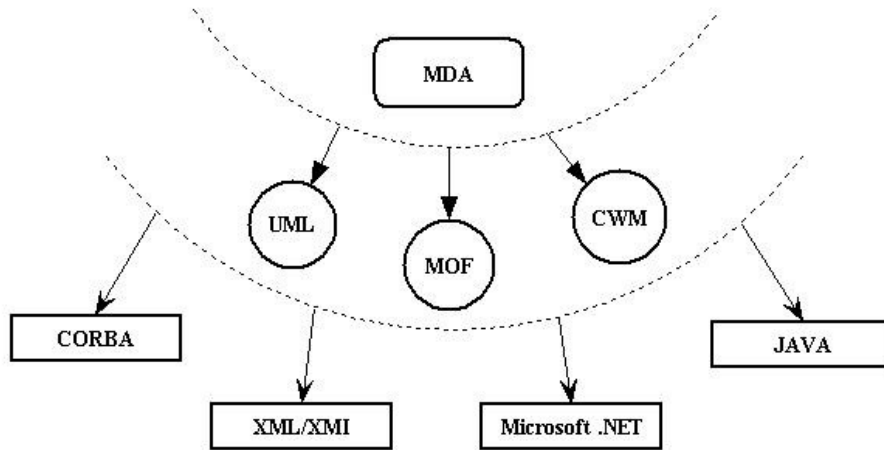


Fig. 1.1. The scheme of MDA interoperability with software technologies.

Fig. 1.1 illustrates the structure of MDA interoperability with various technologies of software development. MDA is located at the central layer of structure, and it "is developed" outside by means of the second layer containing set of basic components listed above - UML, MOF and CWM; and, at last, at the third external layer some well-known at present software platforms of development - CORBA, XML.NET, JAVA - are represented. We shall note that according to OMG basic idea all potential development technologies can and should be placed at this external layer. So, it is the fact that OMG considers MDA as more than just a new technology, but, rather, like a "metatechnology" for applications creation, and henceforth this "metatechnology" is proposed to be solely actual, regardless of appearance and elaboration of new development tools, which have been "integrated" into MDA beforehand.

Reasons for MDA Appearance

In the last decade the process of active development of various information technologies at enterprise level becomes evident. Among these information technologies are the listed above software platforms: Microsoft .NET, CORBA, JAVA, etc. Each of these technologies represents sufficiently complex and volumetric program system containing set of interrelated components, interfaces joined by bulk of special rules and restrictions. And, although each of these technologies generally pretends to provide complex solution of all problems of information systems development, nevertheless, as practice shows, it does not prevent, on the one hand, to co-exist them simultaneously, and, on the other hand, there are no guarantees, that new technologies or development platforms will not appear in the near future. It is obvious for today that availability of great number of platforms and activation of their development process do not promote implementation of these platforms in practice. A main cause of this fact is absence of the consolidated integration mechanism. Heads of large companies at first have to spend significant financial resources for acquisition and development of information systems, their support, training of staff, etc. Subsequently, if

they want to replace the platform, it is necessary to repeat all mentioned stages again spending more resources. As often happens in large companies the necessity occurs to realize simultaneous implementation and operation of several information systems constructed with usage of various development platforms which in addition should interoperate with each other. In this case expenses required for highly skilled staff can exceed acceptable bounds, because specialists experienced in several technologies at once are a rarity, and their "price" increases disproportionately to quantity of software platforms in which they are experts. One can continue to make examples of such situations, but it is already obvious, that the problem of integration should be solved in order to unify all available for today and potential future various technologies of information systems software development. This problem can be solved at least by two ways. The first way – external unification of information interfaces. At present XML language is one of the most commonly used tools of such approach implementation. This way makes the solution of external integration task easier, but it does not provide the solution of systems redesigning problem when changing platforms. OMG consortium proposes another way – "inner" platforms integration by means of platform-independent models.

Application Model. Types of Models.

The basic MDA idea is to separate stage of application operation logic (business-logic) development as independent and mandatory phase. According to MDA concept the application development should be started from modeling stage, that is creation of *application model*, which defines structure and behavior of the future software product.

The model represents set of elements and their links reflecting type of the future software product. It is important, that this model type is generated abstractly without binding to specific programming languages or environments. The model reflects more likely domain, in which designing is conducted, and contains the abstract description of entities: objects, properties, methods and links. In this case the "abstract" means "independent from development platform". At the same time objects themselves can (and should) be described in detail. For example, the model of the application intended for company's personnel administration, can contain objects such as "employee", "department", etc., each of which can be described if necessary in detail by several dozens of attributes (properties or parameters). With all this going on, such detailed description is created not in the programming language, but by means of Unified Modeling Language (UML). Further in this chapter we shall make the acquaintance of UML basics, and now we just shall note, that UML is the platform-independent language created (with direct participation of OMG consortium) specially for application in systems of the object-oriented analysis and designing (OOAD). Now it is necessary to give a more precise definition – up to here the matter concerned the platform-independent models for which PIM (Platform Independent Model) term is used in MDA. It is reasonable that having for an object creation of applications functioning on various platforms, MDA cannot be limited only to PIM-models, i.e. it is necessary "to adapt" abstract PIM-models for some specific environments and platforms of development. Therefore, in addition to PIM-models the MDA-architecture contains concept of platform-dependent models – PSM (Platform Specific Model). These models just play the role of specific "adapters" or "drivers" providing correct mapping of abstract PIM-model onto the concrete software environment. In this case the software environment or a platform mean specific technologies for application creation, for example

Microsoft .NET, Sun One, J2EE, CORBA, etc. It is necessary to note at once, that MDA concept also includes some mechanisms realizing interoperability between PSM, thus providing not only multiplatform property, but also information integration of the applications "generated" for various development platforms.

Development Cycle of MDA Application

In general the development of the MDA applications includes three main stages (see Fig. 1.2). At the first stage the modeler forms PIM model proceeding from the concrete problem. When creating the model the builder abstracts his mind from specific software or hardware program. At the second stage, after PIM creation the modeler develops one or several PSM that serve as "adapters" or "drivers" providing integration of PIM with one or several technologies of software products development.

At the following stage, PIM- and PSM-based application code and, if necessary, the database are generated. If there are several PSM, the procedure of generation can be carried out for each platform being used. The procedure of generation also can be repeated several times without PSM changing, for example, at customer's request (see below). Thus code and databases generation is carried out automatically by means of special software tools.

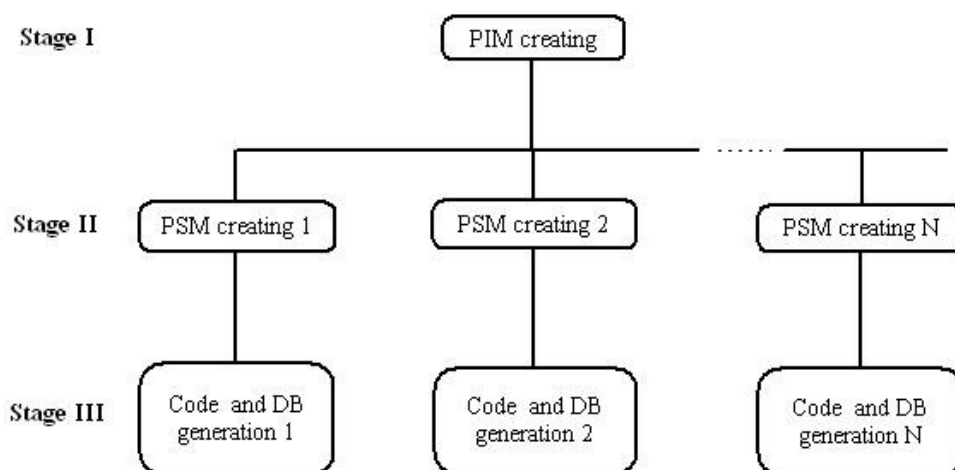


Fig. 1.2. MDA application developing stages

So according to MDA concept the key point of the application development carry over from programming stage to the stage of model creation. At the same time having created the model once, the modeler receives outstanding opportunity of applications generation for different hardware and software platforms.

Shortcomings of the Traditional Approach

Having familiarized briefly with the main ideas incorporated in MDA, let's consider traditional approach of applications development and formulate its shortcomings and constraints. Figure 1.3 shows in general traditional development stages.

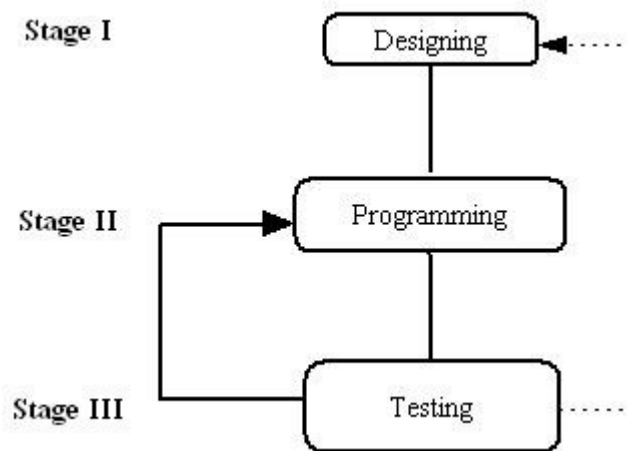


Fig. 1.3. Stages of traditional development

The first stage is the application design creation based on the initial project requirements. The project can have textual or graphic representation with mapping structure of the future application. As a rule, at this stage the developer has already known the type of platform for the future application, therefore the project can include elements and rules specific for the chosen platform. For example, when projecting hierarchy of classes for the application by means of C++ programming language, the developer has a right to lay the foundation of using the possibility of multiple inheritance at the designing stage, and when developing on Delphi other methods can be applied. At the second stage software programming of the application by means of the selected language occurs. At last, the third stage includes debugging and testing. In this case tests also include demonstration of the application to the customer. In practice, in most cases, the development is not over, because the third stage reveals some remarks and errors. Remarks can be both objective, that is caused by incorrectly realized stage of programming, and subjective – in this case the customer or the manager refine the problem statement. Therefore engineering process of complex application is iterative, i.e. includes back-off to the previous stages. And here when using traditional development approach the following stable fact takes place – developers “forget” to come back to the Stage I (dashed line in Fig. 1.3). Instead of application design revision the developers simply modify the code, i.e. return to the Stage II (full line in Fig. 1.3). What is a result? For simple applications there is no matter, the application code is updated, the number of iterations in this case is insignificant, and size of the code is also insignificant. But when dealing with complex program projects especially in case of team operations quite another situation occurs. The “runaway” of the project from actually developed software takes place, i.e. during iterative process the project “diverges” more and more from its implementation path. As a result in many cases the developing process becomes uncontrolled and code modification – more and more difficult, because the project is not documented. If the team of software engineers works, the situation gets complicated, in the absence of documented project the number of errors increases and links between separate parts of the software product break. Whether it is necessary to talk that it is practically impossible to return to such developing process after a time in order to finish it

or to create a new product on its base. Unfortunately, in practice the described situation is not uncommon.

Another constraint at traditional development is necessity of “manual” reprogramming of the application or its redesigning during the platform changing. In spite of the fact that some means of multiplatform development are available, it is impossible to claim with certainty that necessity of “manual” debugging of program code is missing in creation of large software systems.

It is necessary to note that project bad controllability causes such a problem as lack of clear "mutual understanding" between the software developing team and the customer. By the reasons stated above when actually there is no description of the application, the manager and the customer completely lose a control over development process state, because this function implementation requires knowledge of programming languages and other specific knowledge, that is very seldom in practice.

It is also obvious, that complex program systems creation by means of traditional methods, including the stage of “manual” coding demands a lot of time , and development of such applications is extended for years.

We have reviewed only some of the constraints and shortcomings of the traditional approach of applications development. Now it’s time to examine what MDA suggests in this regard.

MDA Advantages

Represented developing cycle of MDA applications (see Fig. 1.2) also contains potential possibility of iterative process. However in this case the developer comes back to stage I and, updates PIM-model of the application, if necessary. And since PSM model and code generator should be developed completely and operate in automatic mode (“hands-off”), PIM over-patching should be implemented in the changed code of the application without distortion; at least, MDA conception declares such intentions. Here we can see some analogy in application program and drivers of the operating system, if application uses standard driver, and the driver operates nominally, then if we change the application code correctly, there will be no unexpected moments in program running as a whole. It is possible to expand analogy a little, if we replace the available driver by the driver of another device (for example, having upgraded a sound-map in a personal computer), and leave an application without changes, then our application should function nominally as before, interoperating now with another device. That is, having created PIM model once, and substituting then "drivers" – PSM, we shall receive our application functioning on completely different platforms. It is evident from aforesaid, what advantages MDA gives. In addition we can list the following useful qualities of the new approach:

- ❑ Cardinal increase of development productivity; as a matter of fact, when MDA using, the process comes to the correct creation of PIM, thus the stage of manual programming is eliminated;

- ❑ Documenting and simplicity of maintenance. In MDA the PIM plays both a role of project, and the main document – the application description. The model contains all data regarding the program in compact PIM format;
- ❑ Centralization of functioning logic. In contrast to the traditional approach where the logic of the application operation "is scattered" on the program code, in MDA it concentrates in one location – in PIM. The application changes its behavior if PIM is changed;
- ❑ Simplification of the development process controllability. From the point of view of customer or manager, availability of platform-independent PIM significantly makes easier understanding of the project as a whole, and control of the development process. PIM does not include the specific features of programming environments, and that's why its elements are clear for customer/manager. As we shall see further, the UML diagrams represented in graphics sort are demonstrable, and they essentially do not require knowledge of programming or theoretical bases of relational databases development.

MDA Status and Perspectives

It is natural, that creation of such technology as MDA, requires plenty of time. At the end of 2001 OMG consortium has issued the "Model Driven Architecture – a Technical Perspective" document, which becomes the first preliminary MDA specification. OMG does not consider MDA as the competitor of existing technologies (CORBA, .NET, J2EE, etc.). MDA is on more high level of development process generalization, allowing to abstract from platforms on stage of PIM creation, to select one or several platforms and to create appropriate set of PSM at the following stage, and, at last, to receive the application functioning on these platforms at stage of code generation. And, as OMG believes, this approach will work not only for present developing technologies, but also for any of the technologies to be created in the future, by means of creating appropriate adapters - PSM - for the technologies.

If OMG plans come true, in the future the script of creating applications can look in such a way: the developer creates PIM, then selects one or several "adapters of platforms" already existent, then "MDA-generator" is initiated and then we get the finished database application with ready graphic interface. If necessary, PIM is changed, and then the procedure of generation is repeated.

Now the aforesaid statements can cause the sound scepticism, however, considering seriousness of OMG intentions and the deserved authority of this organization, and also taking into account, that MDA realization and implementation are now among OMG major strategic goals, these plans will be realized sooner or later.

MDA Concepts

It is necessary to underline, that MDA concept is not completely developed at the present stage, moreover, there are some specific "competing" concepts for its implementation. Two

main approaches can be marked particularly - "interpreter" approach and "generator" approach. What are their essence and difference?

In the "interpreter" concept the main attention is paid to PIM "implementation" into an executable of the application and to access to PIM in runtime. At that, the application "knows" its own model at a stage of execution, and functions according to this model, "interpreting" it in runtime. Thus, for example, there is a fundamental possibility in application runtime to evaluate or to change the status of any model element. At this approach it is possible in principle to work even without the model class code generation, because the description of the model is stored in the specific format inside an executable.

On the contrary, in the concept of "generator" the attention is focused at the most complete and universal implementation of automatic code generation, using optimization of the code on various parameters such as speed, code-size, etc. This approach is rather more similar to CASE systems.

It is difficult to tell now which of the described approaches "wins" as a result. Running a few steps forward, it is possible to note, that Bold for Delphi software product described in this book combines harmonically advantages of the both conceptions.

Probable Consequences of MDA Implementation

It is easily seen, that the "software developer" concept can change its essence significantly when implementing MDA. Shifting accent on model creation the applications development will be fulfilled not programmers, but rather the domain experts. It is possible, that traditional assignment of databases developers and developers of databases applications will be broken. The fact is that, as it will be shown in the following chapters, it is possible now to abstract from specific DBMS knowledge when developing MDA applications, moreover, in many cases there is no necessity to use SQL language, because MDA tools described in this book enable to work at "higher" level (a business-level), where knowledge of the concrete block diagram of a database or structure of the tables fields is not required.

However, programmers-developers will hardly stay without work, as, on the one hand, creation of MDA-toolkit is extremely interesting in itself; and on the other hand, MDA implementation even now saves the programmers themselves from chore, giving it for the most part to the artificial program intelligence – tools of MDA realization.

Unified Modelling Language

MDA, regarding model of the application as of paramount importance, is directly linked to language for such models creation – with Unified Modelling Language.

General Information

Unified Modelling Language (UML) is the main MDA tool for creating the application models. Possessing possibilities of expressive graphics, UML is simultaneously both a tool of the description, and means of documenting development.

At present there is a lot of literature describing UML (see, for example, [2]). Taking into account importance of UML for understanding and MDA usage, we shall describe briefly UML main features and concepts in this chapter.

Necessity for unification of various approaches to the description of business-applications models was the reason for UML occurrence. Last decade of the past century caused the appearance of several dozens of toolkits for models creation, but they have been unmatched, that prevented CASE-means development, and led to difficulties when implementing. At that time CASE-means (Computer Aided Software Engineering) were used basically as universal graphic tools for visual designing relational databases, with possibility of the subsequent automatic generation of a DB.

The first developer of UML is Rational Software company, the author of one of the first CASE-systems – Rational Rose. In 1995 OMG consortium joined in the process of UML standardization, other companies actively start their work at developing the language, and, after several intermediate versions in 1997 UML 1.0 was released.

Now the last version standardized by OMG is UML 1.4, the development of version 2.0 is at the final stage. At present development of UML is made under coordination of OMG which considers the development and promotion of this language as one of the strategic plans.

It is possible to mark the following specific UML features:

- ❑ UML is the language of visual modelling, it provides visual graphic representation of the model in the form of one or several diagrams;
- ❑ UML is not the programming language and does not contain algorithms and operators in the ordinary sense, first of all it is a tool for description;
- ❑ UML, as platform-independent language, abstracts from specificity of concrete programming languages and development tools.

UML was developed as a universal means of object-oriented design of the complex systems, having the evident graphic interface. At the same time problems were set for UML usage as convenient means of documenting development.

UML is based on object-oriented approach, and includes *the class diagram* for the description of model structure. The class diagram is a basis for creation of the application model, and it plays the major role in MDA applications development.

Business-rules

Well thought-out class diagram contains the basic volume of the necessary information about *business-rules* substantially determining correct functioning of the future application.

The “business-rules” term means conditions and constraints imposed by model on the totality of concepts of the environment modelled by the application. Any business-rule can be formulated on the natural language. For example, sentences such as “each employee works only in one company” or “the book should be published by one or more publishers” represent the some business-rules. However, it is obvious, that usage of the natural language in information systems designing results in a number of complexities because of its ambiguities and uncertainty. When creating UML the significant efforts have been undertaken with the purpose of providing maximum flexibility to the developer formulating business-rules, under condition of formal languages limits existence. It explains the appearance of OCL (Object Constrained Language), which has been created specially for replacement of the natural language at the description of conditions and constraints imposed on model elements. OCL plays extremely important role in Borland MDA technologies examined in the book. It will be reviewed in detail in the subsequent chapters. Now we shall only note that formally OCL is a part of UML.

Class Diagram

The UML standard includes 8 types of main diagrams, describing exhaustively structure and behavior of the modelled application. All MDA developing tools described in this book use now only one of these diagrams types – the class diagram. The class diagram is a main source for PIM creation in Borland MDA.

The class diagram is located on a central place in object-oriented analysis and designing of program systems. It describes static structure of the system, i.e. structure of elements (*classes*), structure of their *attributes* and *operations*, and also links between classes (*relationships*). The class diagram does not contain the information about the system developing cycle in the course of time.

Classes

The class concept lays at the heart of the class diagram. Without going into details, we shall consider, that concept of the class is well-known to the developers who use the methodology of object-oriented programming (OOP). The class is represented in UML models as a rectangle generally divided into three parts (see Fig. 1.4).

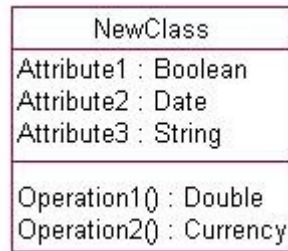


Fig. 1.4. Class representation in UML

At the top the class name is mapped. It is a mandatory parameter. In the middle part *attributes* of the class are mapped, possibly, with the indication of their type and the default value. At the bottom there are *operations*, and, possibly, lists of arguments and the type of returned result. Class can have several attributes and operations. From the programmer point of view attributes and operations are similar to properties and methods in OOP, correspondingly. Let's take as an example the "Employee" class describing the employee of the company (Fig. 1.5).

Here we can see attributes, which names start from the character "/" – "/Age". This character marks derived attributes, which values come out as derivatives from other attributes, and are calculated during application runtime. In the given example the age record can be calculated from attribute «BirthDate» .

The class which cannot have any object, is *the abstract* class. In this case its name is mapped in *italic font*.

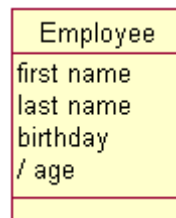


Fig. 1.5. An example of the class with derived attribute

Relationships

Classes in UML can be linked to each other by means of various type relationships.

All possible relationships fall into 4 fundamental types:

- ❑ Dependency;
- ❑ Association;
- ❑ Generalization;
- ❑ Realization

For further consideration it is enough to examine only two of them - *association* and *generalization*.

The association shows a presence of some link between classes, for example, “department – employee”. It is mapped by a solid line (Fig. 1.6).



Fig. 1.6. Association relationship

The line of association can have arrow on one of the ends, pointing the “direction” of the given association. As we shall see further (see Chapter 4), presence of such arrow specifies possibility of "navigation" in the given direction, i.e. overview of class elements values, specified by arrow. If there is no arrow, such scan is possible for both linked classes. The association can have a name, in this case it allocates above a line of association (in Fig. 1.6 name is absent). The ends of an association line are marked by names, which designate roles of the appropriate classes participating in the relationship. In the given example the association has the following roles: “employees” and “Work in”. Besides, the ends of association have dimension marking, which specify a multiplicity of the relationship. So, being guided by Fig. 1.6, it is possible to make the following conclusions about the described model (i.e. to restore business-rules incorporated in it):

- ❑ Each department includes one or several employees (multiplicity “1..n”);
- ❑ Each employee works only in one department (multiplicity “1”);

It is easy to notice quasi analogy between association in UML and relational relationships such as “one – to – many” and “many – to – many” in DBMS. However, it is not possible to carry out this analogy far off. First of all, in a relational DB “many – to – many” link cannot connect two tables, the intermediate (bundle) table is necessary. It is quite usual situation in UML. On the other hand, multiplicity in UML can have, basically, any value including zero. Thus, if we have written “0..n” on the end of the association specifying the employee, it would mean, that there can be the departments without any employee. And if we have set multiplicity “15..50”, it would mean, that number of employees in the department cannot be less than 15 and more than 50 employees.

In some cases the association relationship can link not two classes, but to be applied to the singular class. We shall consider UML model represented in Fig. 1.7.

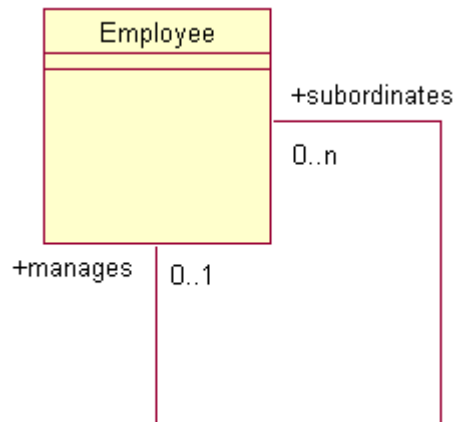


Fig. 1.7. Association with the singular class

In this model both ends of association are linked to the same class, and they determine the following business-rules:

- ❑ Each employee can be subordinate only to one employee (for example, to be subordinate to the manager), or not to be subordinate to any of employees (if he is the manager himself) – multiplicity of relationship $<0 \dots 1>$;
- ❑ Each employee can lead several other employees (if he is the manager), or lead nobody (if he is the ordinary employee) – multiplicity of relationship $<0 \dots n>$.

Use of associations in such variant occurs quite often at creation of applications models.

The relationship of association as particular, but independently existing in UML variant, includes “aggregation” type of the relationship.

The “aggregation” relationship describes situations when classes are linked as “the part and the whole”, i.e. one class is included into another. Such relationship is represented in UML as a line with rhomb on the end, and the rhomb is on the side of “the whole” (Fig. 1.8).



Fig. 1.8. Aggregation relationship

The given type of the relationship shows what components form the concrete class, i.e. realizes decomposition of the whole on constituent elements.

The next type of examined relationship is *generalization*.

This type of relationship specifies that there is a communication between classes such as “parent – child”, it is similar to the inheritance relationship in OOP. Such communication is mapped by a line with an arrow in the form of an empty triangle, and the triangle-arrow specifies a “parent”, i.e. a parental class. In UML the parental class is named “*super class*”.

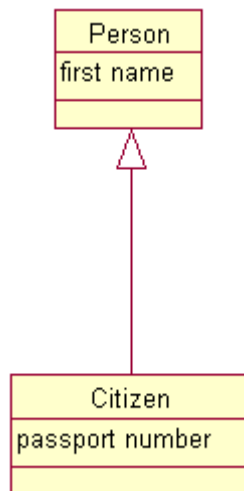


Fig. 1.9. Generalization relationship

Fig. 1.9 shows the relationship of generalization between a parental class “Person” and child class “Citizen”. Thus, it is necessary to take into account, that all attributes of the parental class automatically belong to the child class, though they are not mapped in the child class. Therefore the “Citizen” class will include attribute “name”, besides “Passport number” attribute mapped there. Generalization relationships are frequently used at models building, thus some abstract class quite often serves as the parental class (super-class).

Generalization relationships can form chains of inherited classes, providing the increasing concrete definition of the child classes description based on principle: “from the general – to the particular” (Fig. 1.11).

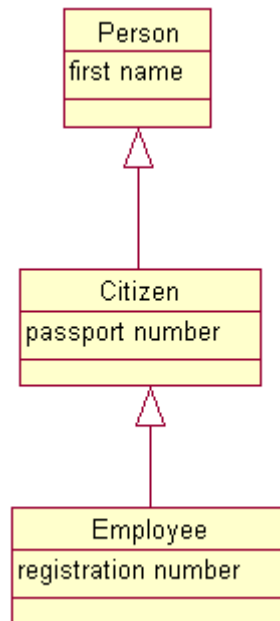


Fig. 1.11. The chain of the generalization relationships

Use of relationships of generalization allows to remove the repeated description of identical properties-attributes of child classes, and to concentrate on their specific properties.

It is necessary to recognize clearly relationships of aggregation and generalization. In case of aggregation the class representing “the part of the whole” is not obliged to have the attributes common with a class representing “the whole”, it is independent in this sense. On the contrary, the child class concerning generalization always possesses a full set of the parental class attributes. If we examine analogy from object-oriented programming, then in case of “aggregation” type association, attribute with “part” class type is included into the structure of attributes of “the whole” class.

Classes-Associations

Classes-associations serve as separate important element of the class diagram. These classes are not “independent”, and they are intended for the additional description of properties of some association, and inseparably linked with it. Let’s examine the model represented in Fig. 1.12.

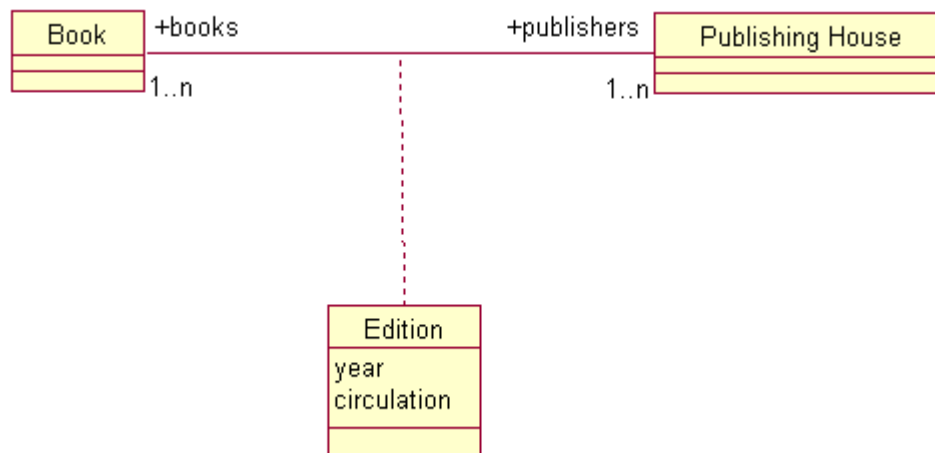


Fig. 1.12. The model including the class association

The model contains two classes – “Book” and “PublishingHouse”, everyone with the set of attributes. These classes are linked to each other by the association, which have <1..n> type multiplicities on both ends, that can be expressed by the following business-rules:

- ❑ The book can be issued by one or several publishing houses;
- ❑ The publishing house can issue one or several books.

Further, we suppose, that the developer has got the following task: to provide retention of information about the publication year and circulation of each book by each publishing house. It is obvious, that this information should be contained in attributes of some class. But which class? Evidently it is impossible to place the specified information in “Book” or “PublishingHouse” classes in the simple way, as the given information depends on both classes, i.e. on concrete pairs of “Book – PublishingHouse” values.

Just for such cases the possibility of classes-associations creation is provided. Having created “Edition” class-association with “Year” and “Circulation” attributes, we solve a problem. For clearer understanding it is possible to carry out analogy with relational DBMS (RDBMS). In essence, the association in this case at RDBMS level is mapped in the bundle table resolving the relationships “many-to-many” between “Book” and “Author” tables. In this bundle table two mandatory fields should be. We shall name them conditionally as “Book_code” and “PublishingHouse_code”. Addition in “Edition” class-association model can then be presented as simple addition of two more fields: “Year” and “Circulation”, to the specified bundle table.

It is important to have in view that the class-association cannot exist separately from the association presented by it, and if association is removed from the model, class-association is removed automatically.

Packages

In UML there is such concept as a *package*. Packages represent the so-called “containers” for the model classes that combine in itself some sets of classes with their relationships, determined by user. Packages are used for grouping of classes with similar purpose, allowing to present the big model containing hundreds of classes, as set of several packages. There is a special identification for UML package (Fig. 1.14), where the name of the package is given, which is chosen by the developer. Each model class can belong only to one package. There is a possibility of packages inclusion into other packages, in this case the enclosed packages are said to be subpackages.

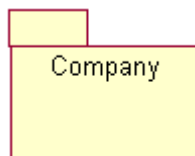


Fig. 1.14. The package identification in UML diagram

UML-model and DB scheme

In this chapter we've used the analogies based on structure of relational databases development. And, though, from the point of view of functional purpose and a scope, probably, it is not absolutely lawful to carry out the comparative analysis of UML classes diagrams development principles and of block diagrams of relational databases, nevertheless it is meaningful to try doing it. It is useful from that point of view, that the most of the developers having traditional experience of databases applications creation, meet some difficulties when turn to MDA-technology. As practice shows, in this situation their “traditional” experience even can hinder.

At first glance, it seems right to recognize the following basic analogies:

- ❑ Class (UML) – Table (PDBMS);
- ❑ Association (UML) – Link (PDBMS).

Let's examine them more in detail. The concepts of a class and a table are similar in many respects, however there are some moments of principle, which don't allow to equate class concept with table concept. First of all, the class, as is known from OOP, has such principal property as inheritance. As we've already known, in UML classes diagram such link between classes is mapped by generalization relationship. There is no analogue in RDBMS. Secondly, in class structure there can be operations that is reflection of another important property of a class in OOP – behavior (presence of class methods). There are also no direct analogues in RDBMS. In RDBMS it is possible to recognize as indirect analogues of some “behavior” such essence as triggers by means of which reactions to the certain events are realized.

Let's turn to associations and links. The first difference, on which we shall draw attention, is the fact that in UML associations link classes, but not single classes attributes.

In RDBMS links are connected to concrete key fields of tables. It is easy to notice, that concepts of key fields, secondary keys and indexes in UML are really absent (though if necessary in some situations is possible to include additional attributes into the class for these purposes). The second important difference –in UML associations can have arbitrary multiplicities of relationships on the ends, for example, “many-to-many” or even “<0..5> to <37..154>”. RDBMS in this case requires presence of the bundle table, and possibilities of multiplicity are constrained by concepts “one-to-many” and “one-to-one”.

Thus, even from brief and incomplete comparison stated above it is obvious, that seeming analogies are not analogies at all. Therefore when developing the model on the basis of the UML classes diagram of it is inexpedient and even harmful to be founded on the block diagram of a relational database. Thus, it is necessary to be based actually on object domain; at that, the UML classes diagram is one of basic tools of its description.

Reasoning from aforesaid the reader can have the natural question: how do database MDA-applications, “breaking” main principles of RDBMS operation, function in that case? Briefly, the answer can be formulated in such a way: in set of tools for Borland MDA realization presented in this book there are the special means carrying out developer-transparent “object-relational mapping”, at which transformation of the classes diagram (model) to RDBMS block diagram takes place. Running forward, it is necessary to note, that at work with Borland MDA the developer practically is not engaged in database creation. Moreover, the developer even may not know the database structure, and may not use SQL language at work. More in detail these questions will be considered in the subsequent chapters.

UML Editors

In conclusion of this chapter we shall give the brief review of the tools providing UML models development. Now there is a lot of such tools (UML-editors). Rational Rose software product created in 1998 is the very first of such tools. It is still urgent, and is actively applied by many developers up to the present time. Rational Rose represents universal CASE-tool for applications modelling and development, having extremely advanced program interfaces with various languages and programming environments. Rational Software company, facing at sources of UML creation, is developer of Rational Rose.

NOTE

Rational Software company was purchased by IBM in 2002.

Development Principles in Rational Rose

Here we shall very briefly get acquainted to principles of work and capabilities of Rational Rose editor. The reader can find fuller information, for example, in the book [2].

Actually, Rational Rose UML editor is the standard among UML modelling tools. It possesses a full tool set for creation of all basic diagrams providing by UML language standards. In Fig. 1.15 the general view of Rational Rose program interface is represented.

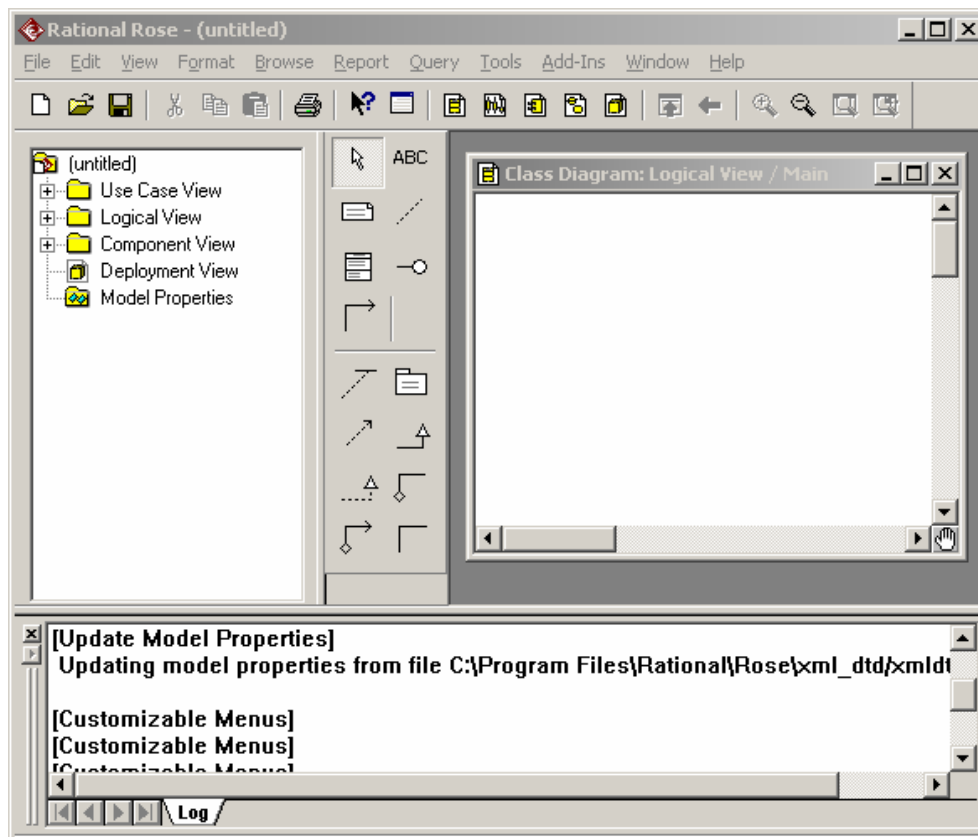


Fig. 1.15. The general view of Rational Rose interface

Rational Rose gives the developer extensive capabilities for model setup. After new UML model creation (main menu items: File-> New) for its parameters setup you can choose corresponding items of Tools menu (Fig. 1.16).

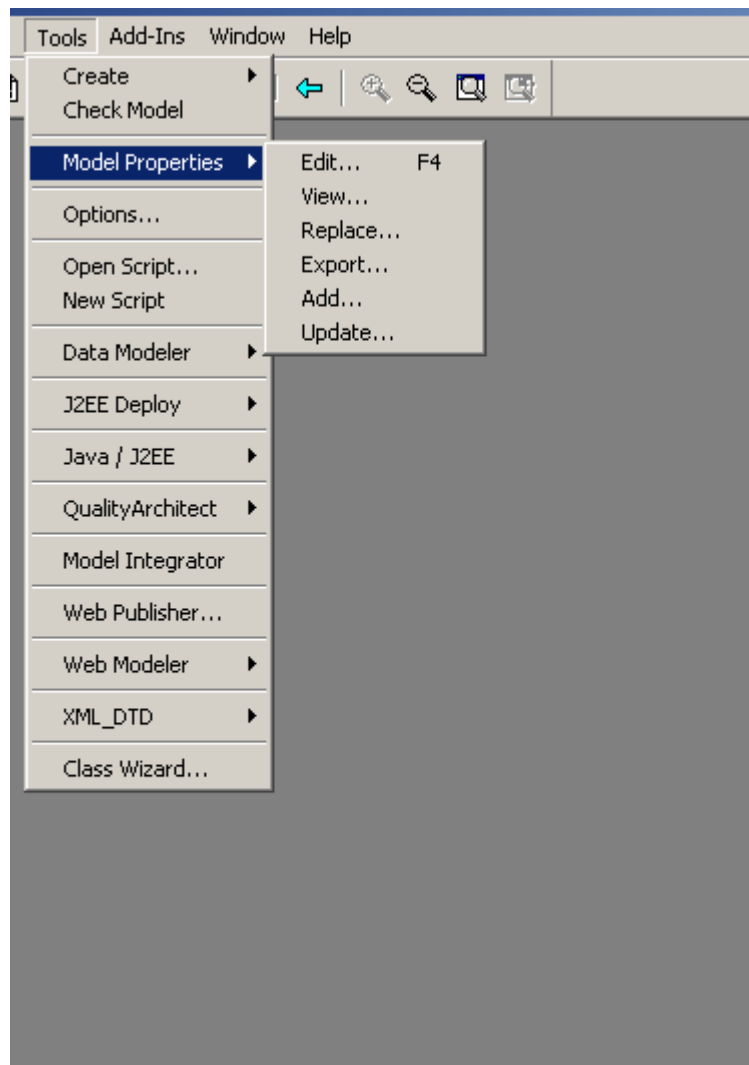


Fig. 1.16. Menu of model parameters setup in Rational Rose

After choosing “Edit...” menu item (to edit the model properties) the main window of UML model properties editing (see Fig. 1.17) will appear. For the beginning we shall set the toolbar for UML model. This panel includes the visual elements used at models creation (classes, packages, associations). For editing the panel type we shall press the button with dots from the right of UML item (Fig. 1.17), and we shall get into the editor of visual elements set (Fig. 1.18).

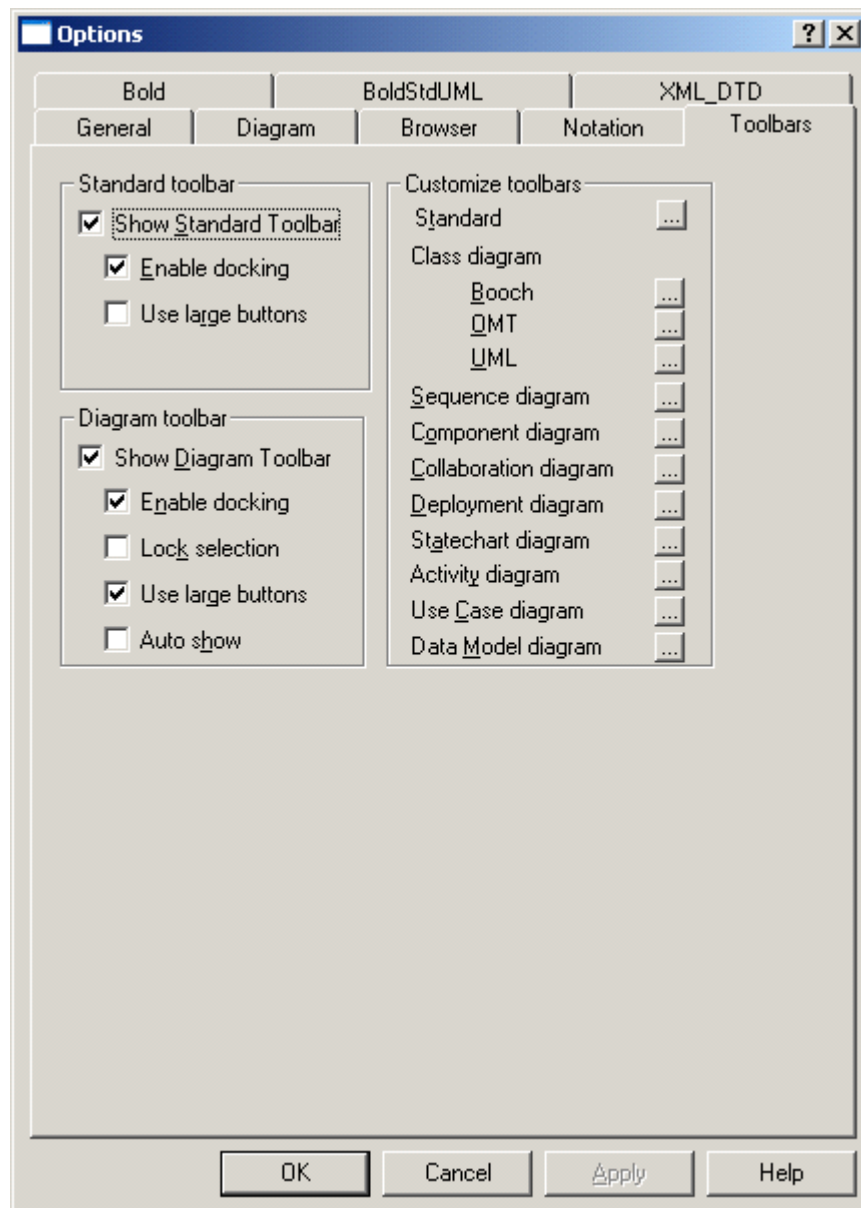


Fig. 1.17. The main window of model properties setup in Rational Rose

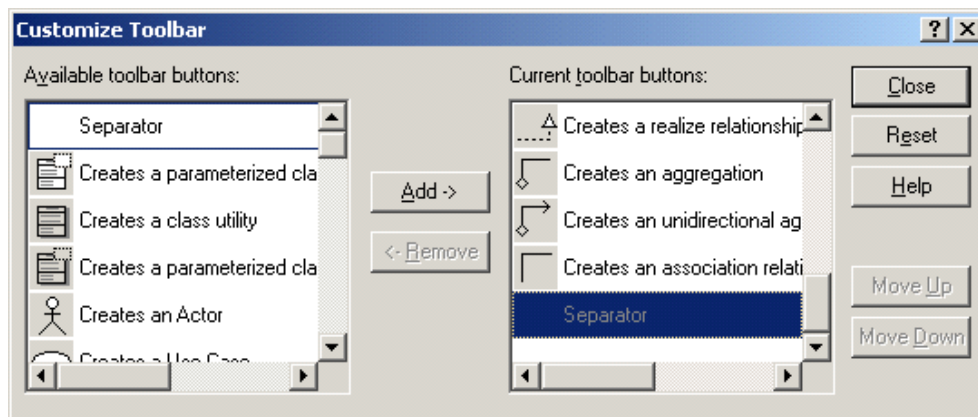


Fig. 1.18. The editor of visual elements set of Rational Rose

This editor is executed by rather traditional way, allowing to transfer necessary elements between a full set of all elements (the left window) and the current set of the user (the right window). Thus, we shall generate the type of toolbar sufficient for most of development variants (see Fig. 1.19).



Fig. 1.19. The adjusted toolbar

Use of this panel for model creation is not troublesome. For addition in model of some element, for example, an element of a new class, it is necessary to choose it in the toolbar, and then to click on a free field of model, thus the new class with parameters by default will be displayed. For individual class setup it is enough to call the pop-up menu for setup type selection by clicking the mouse right button.

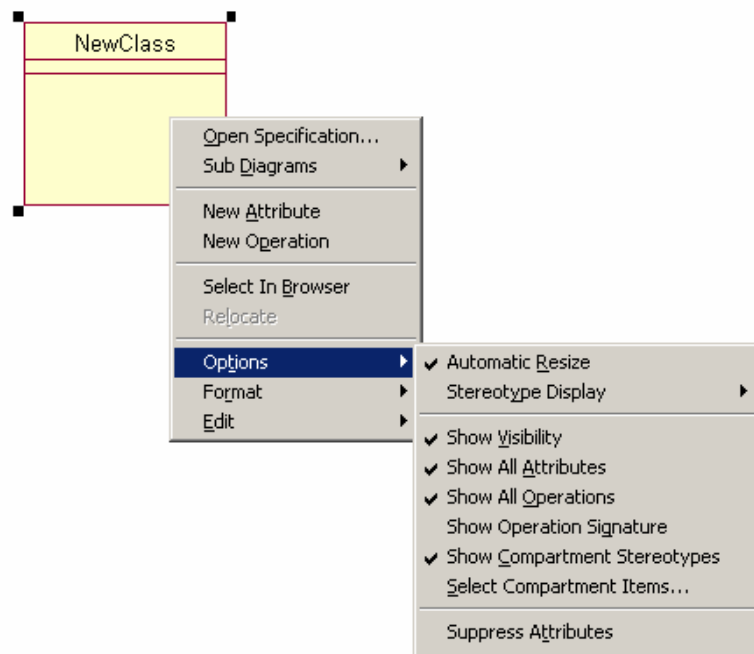


Fig. 1.20. The pop-up menu of class parameters setup

At that consistently enclosed pop-up menus (see Fig. 1.20) will appear, containing numerous capabilities for class parameters setup. So it is possible to set class visual parameters, visibility of attributes, operations, parameters of automatic adjustment of the class image sizes, etc. For fast jump to class adjustments it is necessary to click twice on its image. In result the main window of class parameters setting will be displayed (see Fig. 1.21).

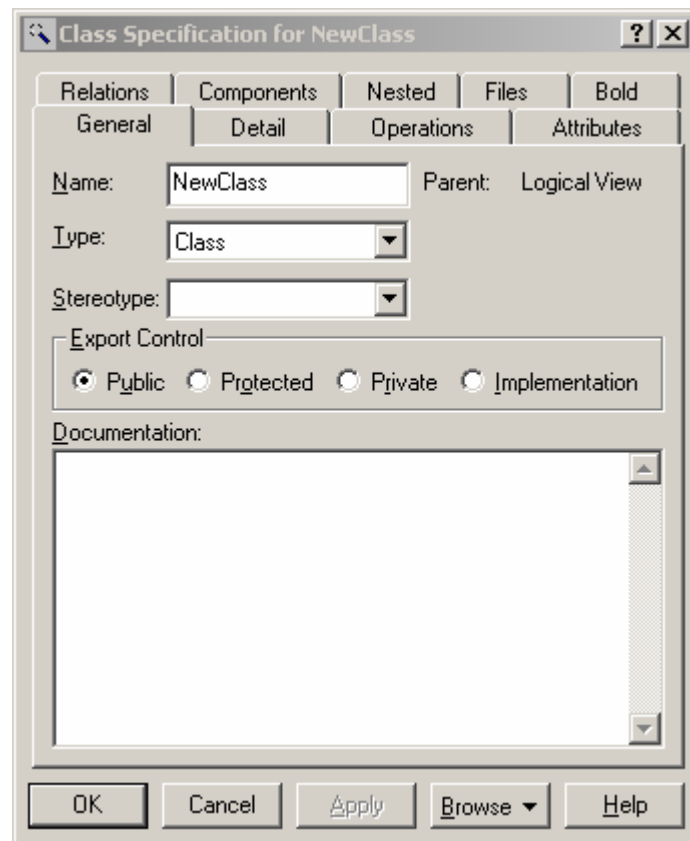


Fig. 1.21. The main window of the class parameters setup

This window has several bookmarks. In “General” bookmark it is possible to set class name, type and some other parameters. Besides here in the special window “Documentation” the developer can write the additional text descriptive information intended for the purposes of documenting.

In “Detail” bookmark it is possible to set the detailed information for model, whether the class in DB is persistent, or transient, or abstract, etc. (see Fig. 1.22).

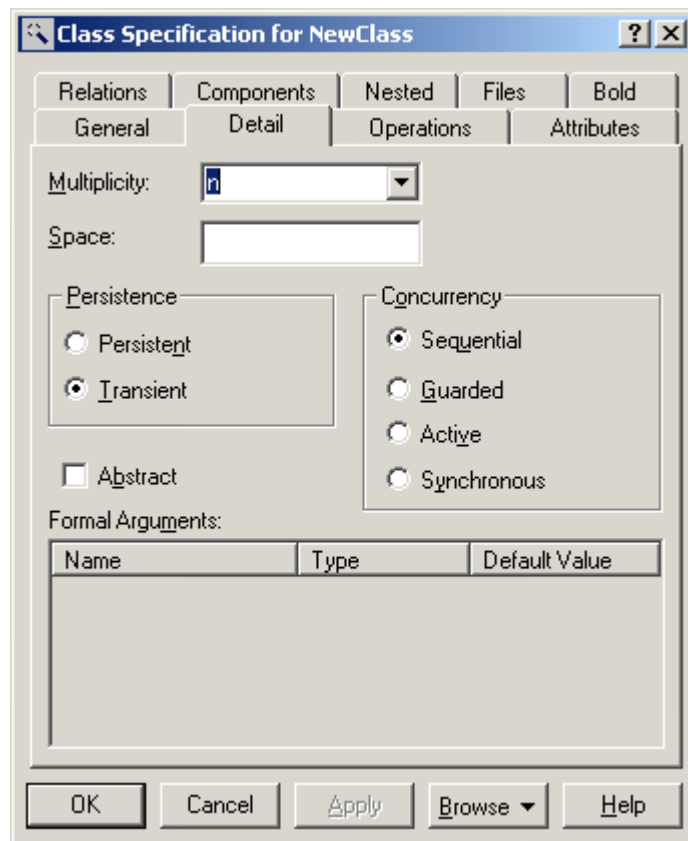


Fig. 1.22. Detailed setting of the class parameters

Using the described capabilities, we shall create simple model represented in Fig. 1.23, which contains two classes, linked by association. For association creation it is necessary to select its element in the toolbar, then click on the first class and draw a line up to the second class.



Fig. 1.23. View of unadjusted model in Rational Rose

Now we shall briefly describe procedure of association adjustment. Having click twice on association line, we shall open the main window of its parameters setting (see Fig. 1.24). This window also contains several bookmarks. In “General” bookmark it is possible to set association name and names of roles for each class.

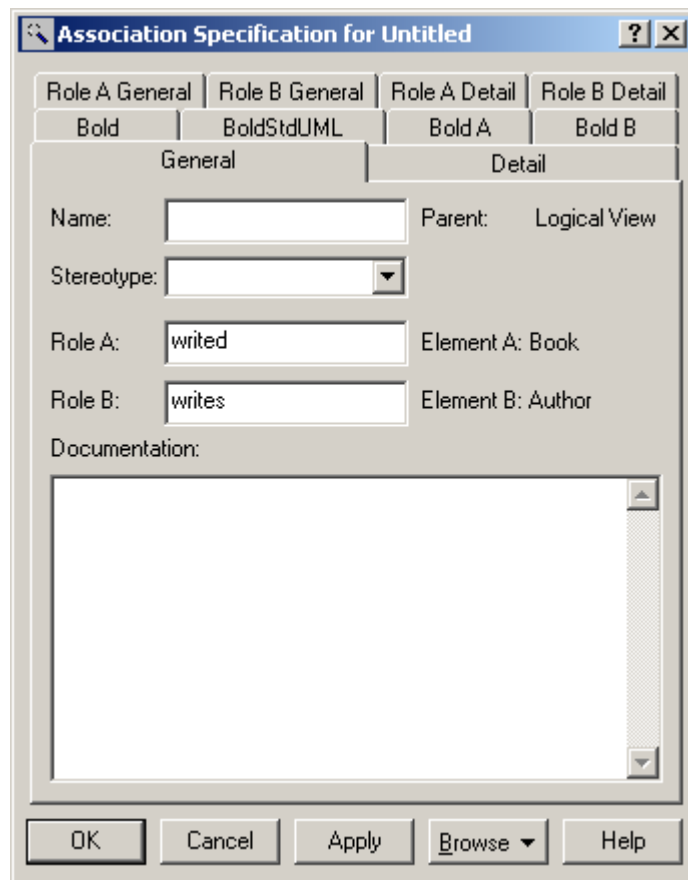


Fig. 1.24. The main window of the association parameters setup

In “Role A Detail” and “Role B Detail” bookmarks it is possible to adjust detailed parameters for each association role (see Fig. 1.25), for example, to set names of roles and multiplicities of roles.

Fig. 1.25. Detailed setting of the association parameters

After carrying out of the specified adjustments our model will look like you can see in Fig. 1.26.



Fig. 1.26. View of adjusted model

Other UML-Editors

Almost all up-to-date CASE-systems possess built-in UML editors. For example, PowerBuilder software product of PowerSoft Company also provides high functionality for UML-diagrams creation. Starting with Delphi 6 version the capability of integrating ModelMaker UML-editor into this development environment has appeared. This program

tool also allows creating complex class diagrams, and besides it possesses the advanced capabilities on interoperability with a code of application developed in Delphi.

In structure of the newest Borland development environments examined in the appendix to this book, home means of creating UML-diagrams, which are closely integrated with development environment, are included. It is necessary to note, that with the advent of XMI language standard (XML Metadata Interchange), the basic opportunity of UML-model development in any editor supporting XMI specification, with an opportunity of the subsequent editing this model in the other editor, or use of this model in MDA-applications development environments, for example, in Borland MDA program tools, has appeared.

Summary

In this chapter the brief review of the basic concepts of new technology of applications development – Model Driven Architecture (MDA), and its comparison with traditional approaches are given, and also the brief information on unified modeling language – UML, is represented.

Prerequisite of MDA appearance is a plenty of difficult-to-integrate development program platforms, each of which is complex and expensive in development, implementation and maintenance program system.

At the heart of MDA the principle lays of creating application platform-independent model (PIM), which determines structure and behavior of the future software product.

There are also platform-dependent models (PSM) playing a role of adapters for various platforms of development.

Presence PIM and PSM models allows to generate automatically a code of the application and, if necessary, a database.

Changing PSM allows receiving automatically functionally identical application for other platform without change of the user code.

It is shown, that MDA use has the important advantages over the traditional approach of applications development – by productivity, transferability, controllability, etc.

MDA technology is in process of development, and there are some concepts of its realization.

MDA implementation should lead to significant changes in a role and a place of programmers-developers, experts in DBMS sphere, etc.

MDA is based on unified modelling language (UML).

UML is platform-independent language, and it is intended for models creation.

UML is not the programming language, but it represents the language of the graphic description that is realized by means of diagrams.

The class diagram included in UML is intended for the description of static structure and composition of model. The class diagram is the basic information provider concerning

business-rules, and it is the unique UML diagram used in Borland MDA toolkit for PIM-model formation.

Basic elements of the class diagram are classes and relationships between classes. There are special classes-association for realization of relationships additional properties storage.

As a result of the brief comparative analysis of the UML class diagram and structure of relational databases the basic conclusion is made about inexpediency of analogies carrying out between these concepts.

Chapter 2. Borland MDA Review

This chapter represents the review of software tools and technologies developed by Borland company for MDA-applications realization.

What is Borland MDA (BMDA)

In this book under “Borland MDA” joint name, ideology, technologies and toolkits are collected, which are included in software products of Borland company, and meant for practical MDA realization at the applications development. The “Borland MDA” term has appeared at the end of 2002 when Borland company had been purchased several firms-developers of software engaged in creation of toolkit for MDA elements implementation into the development process. In the following section fuller information will be represented. Further in the book we alongside with this word-combination shall apply the abbreviated term: “BMDA”.

Let's note, that alongside with the “Borland MDA” term in some sources, including Borland company's ones, other similar term is used as well: “Model Driven Development”. However, in author opinion, such interpretation is a little “narrow”, because the role of Borland MDA tools described in this book is not limited only to applications development cycle. If it was so Borland MDA tools examined in this book should be related to CASE-systems class. But it does not meet the real facts. As it will be shown further, Borland MDA technology and tools are of great importance at all stages of applications creation, including a stage of program execution.

In Fig. 2.1 conditional general scheme illustrating the structure of the elements forming set of Borland MDA program means is presented. It includes:

- ❑ Tools of designing and modeling: are intended for creation of the application model. UML-editors, means of checking UML-model integrity and consistency, editors of OCL language, and tools of models import/export are included;
- ❑ Development tools: provide object space program realization, interfaces with presentation layer and persistence layer. Classes and components used by the developer when creating applications are included.
- ❑ Generation tools: provide generation of an application code in base language (in Delphi – Object Pascal). The generated code represents program realization of all model elements in base language. Besides these means also provide generation of relational databases schemes.
- ❑ Debugging aids: are intended for debugging MDA-applications. Such applications require debugging at a level of objects and messages of object space.
- ❑ Integration tools: provide integration with development environment, on the one hand, and necessary link settings with a persistence layer (DBMS), on the other hand.
- ❑ Interpretation and controlling means: provide functioning at a stage of execution, including interpretation and access to model elements, the control of object space

integrity, the control of interoperability with persistence layers and GUI at a stage of application execution.

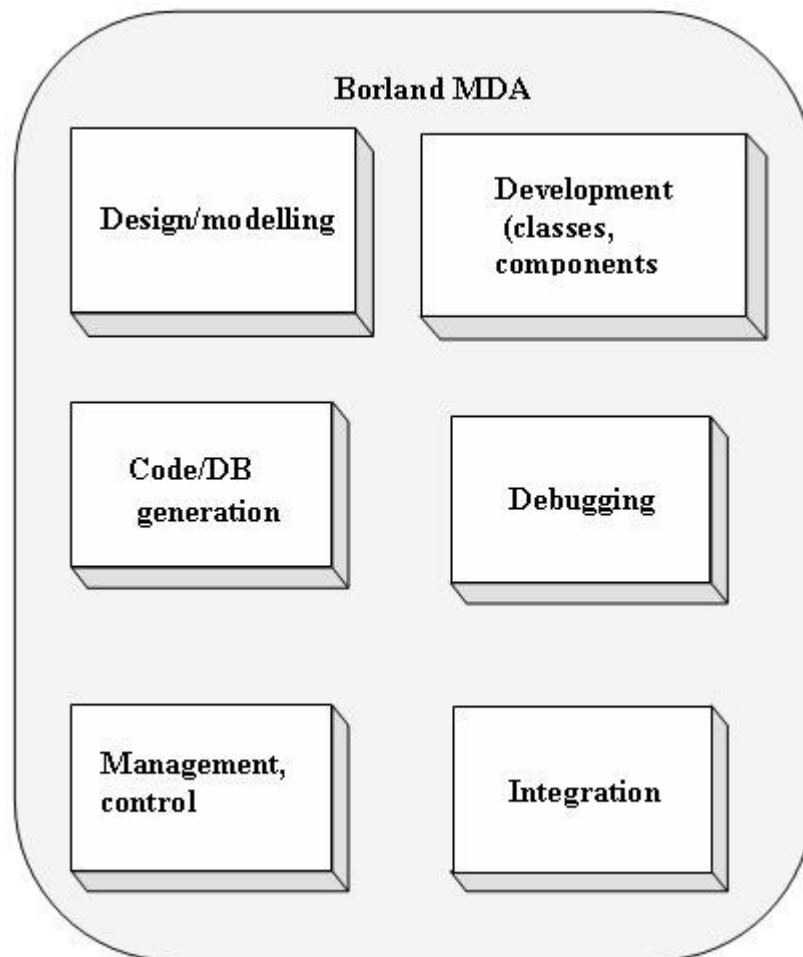


Fig. 2.1. Borland MDA structure

In this book at Borland MDA examination we shall not go beyond Delphi 7 Architect Studio programming environment, however by now this technology (essentially advanced) is introduced into one more software product - Borland C#Builder, and in the near future it most likely will be integrated into other development environments of Borland company.

Development History

Fig. 2.2 illustrates stages of Borland MDA technology development.

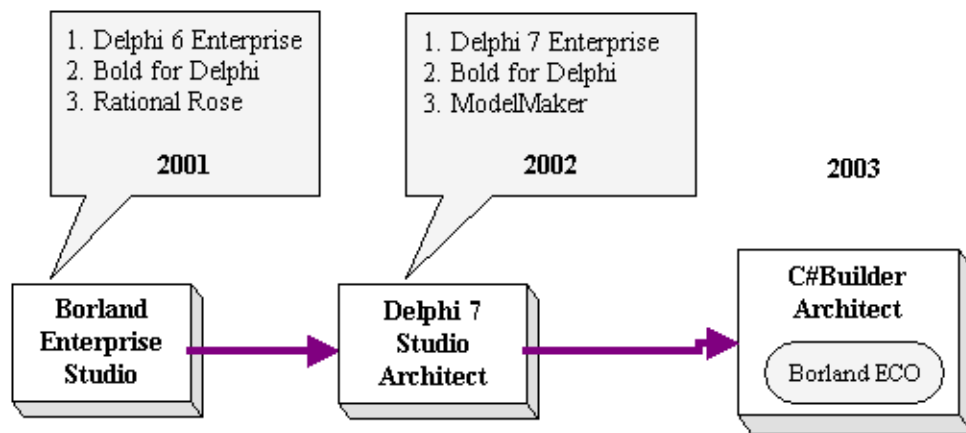


Fig. 2.2. Developing Borland products, supporting MDA

Borland Enterprise Studio

Borland Enterprise Studio program complex, released in 2001, can be considered to be the first product of Borland company supporting MDA technology. The delivery set structure included the following basic software products:

- ❑ Borland Delphi 6 Enterprise (Borland company);
- ❑ Bold for Delphi version 3 (BoldSoft firm);
- ❑ Rational Rose (Rational Software company)

NOTE

Besides software products mentioned above Borland Enterprise Studio included some other software products (Borland Application Server, etc.), however they are not interesting in the context of our description.

According to Borland the basic idea of using such tools composition, was the following: the developer created the application model in Rational Rose CASE-system in UML; then, at the following stage, by means of Bold for Delphi components package, this model “was translated” into Delphi 6 development environment, where the final “operational development” of the application and the user interface development took place. This scheme is very simplified, and actually, in practice this process could look otherwise. Further in this book we shall more in detail get acquainted to this technology of MDA-applications creation using Bold for Delphi product (Delphi 7 Architect Studio) as an

example. And as to Borland Enterprise Studio as a whole, the probable reason of this product and proposed approaches poor propagation was, on the one hand, Bold for Delphi version 3 flaw design, and, on the other hand, in spite of unconditional merited authority of Rational Rose, its capabilities were obviously used far from being to the full, having in mind wide set of its software products. As a matter of fact, Rational Rose was used here only as UML-editor. In other words, Borland Enterprise Studio actually represented a set of the products (created by different developers) relatively poorly integrated among themselves. And, though in itself the idea of the total development process coordination in a single complex was certainly advanced and correct, but its realization for that moment “lagged behind” ideology.

Borland Delphi 7 Studio Architect

At the end of 2002 Borland Delphi 7 Studio Architect version has been released. Its structure, besides Delphi 7, included essentially advanced Bold for Delphi version 4, and, in addition, Model Maker UML-editor. When this version appeared, it becomes possible to create UML-models directly in the development environment, as Model Maker is integrated with Delphi closely enough. Thus, necessity of involving third-party UML-editors, such as Rational Rose, has disappeared.

On the other hand, approaches and decisions proposed by BoldSoft firm in this Bold for Delphi version, proved to be so successful, that a month later after the described Delphi 7 Studio version release, Borland company has purchased BoldSoft firm.

NOTE

Practically at the same time Borland company has also purchased TogetherSoft and StarBase firms.

After that events Bold for Delphi product carried over to Borland company. It is appropriate to say here a few words about BoldSoft firm. Being a member of OMG consortium, and the direct participant of OCL development, Swedish company BoldSoft MDE AktieBolag for the first time has proposed the complete concept of MDA practical realization in the applications created in Delphi and C ++ Builder, and has developed for these purposes software products: Bold for Delphi and Bold for C ++ Builder. And it is necessary to emphasize, that Bold for Delphi first version created for Delphi 5 has appeared long before OMG consortium accepted the concept of MDA development. Thus, BoldSoft firm did not wait till OMG elaborate rules and recommendations finally, and, even before corresponding standards release, BoldSoft realized in practice its own MDA vision. Bold for Delphi proved to be rather successful product. Suffice it to say, that it has been successfully used in information system of Sweden parliament during several years, and also in some other organizations. In developers' opinion, this product implementation has allowed to increase efficiency and to reduce terms of software systems creation by a factor of 10. Close enough mutual integration of Delphi 7 Studio Architect software products, plus essentially advanced versions of the products themselves, gave such a result. After appearance of this Delphi version Borland company is said to be not only RAD-environments developer (i.e. environments of fast software products development – Rapid Application Development), but also the developer of the products supporting a full cycle of applications creation – from design stage up to the stage of code generation (ALM - Application LifeCycle Management).

C#Builder Architect

And, at last, at the end of 2003 absolutely new product of Borland company for Microsoft .NET platform has appeared, it was C#Builder Architect. New quality of this product consists not only in its capability to support promising .NET platform. From MDA-architecture point of view C#Builder Architect has essentially new aspect – after its appearance we can assert, that integration process of MDA-tools and Borland development environments has been essentially completed. Borland MDA technology named in C#Builder Architect as Borland ECO (Enterprise Core Objects) now is completely integrated into development environment including graphic UML-editor, and also necessary components and classes for MDA program support. In the appendix to this book we shall give the brief review of C#Builder Architect and Borland ECO technologies capabilities.

Capabilities and Specific Features

For various software products of Borland company concrete sets of BMDA technology capabilities differ strongly enough. Here we shall get acquainted to the basic capabilities common for all Borland company products supporting MDA-applications development. Among them there are the following capabilities:

- ❑ UML-models creation; for these purposes built-in UML-editor (text or graphic) is used;
- ❑ Program code automatic generation in Object Pascal language;
- ❑ Automatic generation of relational databases structure; the database structure (scheme) is generated on the basis of the created model;
- ❑ Support of database modification with saving information (DataBase Evolution); the special toolkit allows to change (naturally, in acceptable bounds, and depending on DBMS chosen) the database scheme without losing available data;
- ❑ Capability of database storage in XML-document without DBMS use;
- ❑ Using OCL (Object Constraint Language) as the basic means of information interface formation between different levels of application (DBMS, business layer, GUI).

A matter of principle when using BMDA is the three-level scheme of the application creation, which includes the persistence layer, business layer, and GUI. And in this case it is not abstraction, it is reality embodied in concrete sets of BMDA components, with which the developer deals. So, if usually when creating databases applications in Delphi, visual components (grids, editing windows, etc.) are linked to DB fields or tables, then at work with BMDA, all of them are linked to the intermediate layer – to the objects of business layer. The application business layer formation is one of Borland MDA basic functions. **Other basic function is ensuring interoperability between business layer and persistence layer (DBMS) – object-relational mapping and interoperability.** This interoperability includes automatic, transparent for the developer, OCL translation into

SQL operators, operations with DB tables execution, etc. More in detail it will be described in the subsequent chapters.

And, at last, here is Borland MDA basic difference from CASE-means mentioned above: BMDA functions not only at application development cycle stage, but also at the stage of its execution. Any CASE-means, no matter how perfect it may be, is intended only for design and modelling stages realization. It, certainly, can include also capabilities of code generation and database generation. But after application start CASE-system does not function any more, as a matter of fact, it is not “present” already. BMDA functioning radically differs. Saving application model in the executable, BMDA at the stage of the application execution uses this model with the purpose of business layer control, the control of object space integrity, management of business layer interoperability with persistence layer and GUI. It is necessary to note here, that this BMDA feature is specific enough, and distinguishing this technology from the others. In chapter 1 various concepts of MDA realization were examined. If we conform to the above terminology, BMDA ability to keep “knowledge” of model at a stage of execution allows attributing this technology as “interpreters”. However it is not absolutely true, because BMDA also has got an ability of code generation. But BMDA is not also the code generator. Further it will be shown, that, when using BMDA certain versions (in particular, Bold for Delphi) in some cases generally it is not necessary to generate code of model classes. In the following chapter we shall create the databases application, where in general both model classes code, and the user code will be absent. Running forward a little, we shall explain, that BMDA saves the information on model not in a generated code, but in a special component.

Summarizing the aforesaid, we shall try to define Borland MDA technology in the following way:

Borland MDA is, on the one hand, the technology and the development environment allowing to form object space (business layer) and to realize business-logic of the application at a stage of creation, and, on the other hand, it is the program system providing functioning business layer and its integration with DBMS (persistence layer) and GUI at a stage of execution.

Advantages for Developers

What advantages are given to application developers with using Borland MDA technology? Here we shall list again only the basic advantages of this technology:

- ❑ The unified approach to all stages of development cycle – from designing model to application development. The essence of this approach is that the developer at all stages works with the same entities – model objects. There is no break between the beautiful scheme-model and programming of DBMS application as the developer does not “go down” to the database level, he even can know nothing about DB structure and tables presented there. The developer always uses those names and entities, which have been included in the model by himself;
- ❑ The stage of “manual” creation of database is completely eliminated. All tables, fields, indexes, keys are generated automatically according to the model. To use the concrete

DBMS it is enough to connect and setup one of adapters of the databases included into BMDA. There is a possibility of creating proper adapters of databases;

- ❑ Database modification turns into trivial process – after entering necessary changes into the model it is simply enough to generate a new database; it becomes absolutely unimportant what DBMS to use – when changing DBMS the application and its code do not change;
- ❑ Using OCL allows abstracting completely from SQL-dialect of concrete DBMS. SQL in some cases becomes practically unnecessary and is used fairly seldom although the possibility of its involvement still remains.

It is necessary to note that means of automatic generation of databases and even applications classes existed earlier too. Suffice it to cite as an example Rational Rose and PowerBuilder products of Powersoft company. There is also such software product, as Delphi RoseLink of Ensemble Systems company, being like “a bridge” between CASE-system of Rational Rose and Delphi. Its basic functions are code generation in Object Pascal and reverse engineering. However, the generated code does not contain realization of functionality. Only descriptions-definitions of classes, interfaces, etc. are generated.

Borland MDA goes beyond, as being integrated into Borland Delphi environment, it provides the developer with full set of visual and not visual components, sufficient for realization of application Object Space. Therefore the developer receives an opportunity to work not at a layer of code and tables of DB, but at a layer of objects inside this object space.

In other words, using Borland MDA, the developer:

- ❑ Does not create a database, but forms the application model in UML;
- ❑ Works not with tables, fields and keys of a database, but with objects of the application model created by him – classes and their attributes;
- ❑ Connects visual components for the data mapping not to DB tables, but to model objects;
- ❑ Does not write requests in SQL, but forms statements in a flexible and powerful dialect of OCL.

As it is accepted to speak, in this case development is carried out at *business-layer*.

All aforesaid will be evidently shown with use of concrete examples in the subsequent chapters.

Sequence of Studying

Further in this book practical use of Borland MDA technology by the example of Delphi 7 Studio Architect and Bold for Delphi will be considered. Fundamentals and essential principles of BMDA technology were laid and then have been developed just in Delphi 7

Studio Architect. For the best understanding of ideology and practical development of ECO (Enterprise Core Objects) technology realized in C#Builder Architect it is rather useful to familiarize preliminary with "sources" of this technology incorporated in Delphi 7. It is expedient for the following reasons – first of all, in ECO in many respects there is a certain continuity of principles and approaches. Secondly, in current C#Builder Architect version ECO capabilities in some cases are more limited in comparison with those available in Delphi 7 Architect. It is caused by incomplete at present Bold for Delphi “translation” on new Microsoft .NET program platform. Therefore it is quite probable, that in the future versions and updates of Borland Company software products ECO technology also will be updated and supplemented, and, most likely, it will be made also with the purpose of ECO providing with those properties which were present in Bold for Delphi. Hence, from this point of view it is useful to familiarize with “sources” of this technology too.

And, at last, it is necessary to consider that important fact, that **Delphi 7 Studio Architect is a unique Borland product for today, providing MDA-applications development for Windows (Win32) platform.** If Borland Company continue in the future Delphi “line” for Win32 platform, then, certainly, in these future Delphi versions Borland MDA technology also will be kept and expanded.

Probable Difficulties

Also it is necessary to note beforehand, that on a way of practical mastering by Borland MDA technology, the developer can meet some difficulties.

First of all, Borland MDA is a complex and volumetric program system. For example, program realization of Bold for Delphi version examined in this book includes thousand new classes, attributes and methods. Secondly, at present the technical information on Borland MDA is insufficient. Practically a unique useful source of the concrete information on technology at the present writing are the Internet-conferences [at Borland news server \(<forums.borland.com>\)](http://forums.borland.com), where two basic groups of news on Borland MDA – *“borland.public.delphi.modeldrivenarchitecture.general”* and *“borland.public.delphi.modeldrivenarchitecture.thirdparty”* are placed. And, at last, thirdly, Borland MDA is qualitatively new technology of development, it is possible to say, that it is the whole new world, and when turning to this technology the ideology vision of the developer should be reconstructed rather sharply. All is unusual here from the point of view of traditional methods and means. Therefore in practice rather paradoxical situation is quite probable – the less traditional experience of databases applications creation the developer has, the easier he becomes proficient in Borland MDA.

These cautions, certainly, do not mean, that this technology is very difficult for mastering, actually, hundreds developers in the world relatively for a long time and successfully have already applied Borland MDA in their program development.

Bold for Delphi as Borland MDA Basis

As already it was stated earlier in this chapter, Bold for Delphi software product created by BoldSoft Swedish Company, now is the property of Borland Company. Early Bold for

Delphi versions developed for Delphi 5 and Delphi 6, now are inaccessible to developers and consequently now a unique product of MDA realization in Delphi is Borland Delphi 7 Studio Architect (the opportunity of Borland Delphi 7 Studio Enterprise version updating also exists). All current and the subsequent Borland MDA updating are carried out by Borland Company (in 2003 Bold for Delphi 4.0.0.21 last updated version accessible to the authorized users was released).

What does Bold for Delphi represent? From the point of view of operation by the developer it is the package of components being installed separately from actually Delphi development environment. Bold for Delphi includes more than 100 visual and not visual components. Package “interiors” contain about 1700 classes.

Bold for Delphi basic capabilities are listed below.

- ❑ Built-in textual UML-editor for models creation;
- ❑ Close integration with Rational Rose CASE-system regarding UML-models creation, adjustment, import and export;
- ❑ Supporting XMI language regarding UML-models import and export;
- ❑ Built-in OCL-expressions intellectual editor;
- ❑ Automatic generation of model classes program code in Object Pascal language;
- ❑ Opportunity of creating valuable applications without model classes code generation;
- ❑ Automatic generation of the map for the relational databases accessible through program interfaces BDE, ADO, DBExpress, SQLDirect; Interbase/Firebird DBMS and DBISAM DBMS are also supported;
- ❑ Opportunity of work with user (“non-standard”) DBMS;
- ❑ Opportunity of database storage in XML-document without using DBMS;
- ❑ Support of database map updating with information retention (DataBase Evolution);
- ❑ Using OCL (Object Constraint Language) for realization of flexible and powerful queries to object space, and also for interaction between various application layers (DBMS, business-layer, GUI);
- ❑ The adjustable mechanism of object-relational mapping including automatic translation of OCL operator into SQL one;
- ❑ The mechanism of “subscription” to the events arising in system;
- ❑ Automatically generated graphic forms for mapping and editing the data;
- ❑ Creation of multilink applications and “thin” clients in DCOM base;
- ❑ Native means of MDA-applications debugging.

For cardinal increase of databases applications development efficiency it is enough even basic opportunities listed above. Also it is necessary to note, that using Bold for Delphi is especially “profitable” when by developing applications for the “large” databases including

500 and more tables. Though this product can be successfully used both for local DBMS and for independent databases, where its use also allows to reduce the development cycle in a few times.

Such unique capabilities of Bold for Delphi product could not pass unnoticed by the third-side companies. Now there are some program packages of the third-side software producing companies, which were created specially for Bold for Delphi technology. The brief review of these products will be given in Chapter 15.

Installation and Tools Review

Last version of Delphi 7 Studio Architect product (trial edition for acquaintance) is accessible to loading from Borland Company site. At Delphi 7 Studio Architect Trial Edition installation the panel (see Fig. 2.3) will appear, allowing to select products to install: Delphi 7, ModelMaker and Bold for Delphi.



Fig. 2.3. Panel for Delphi 7 Studio Architect trial edition installation

Products are installed in order of their enumeration. After Delphi 7 installation it is recommended to install Model Maker UML-editor as well, allowing to create UML classes diagrams (though it's not of necessity). If at Bold installation it is proposed to select composition of installation (see Fig. 2.4), it is recommended to mark all components.

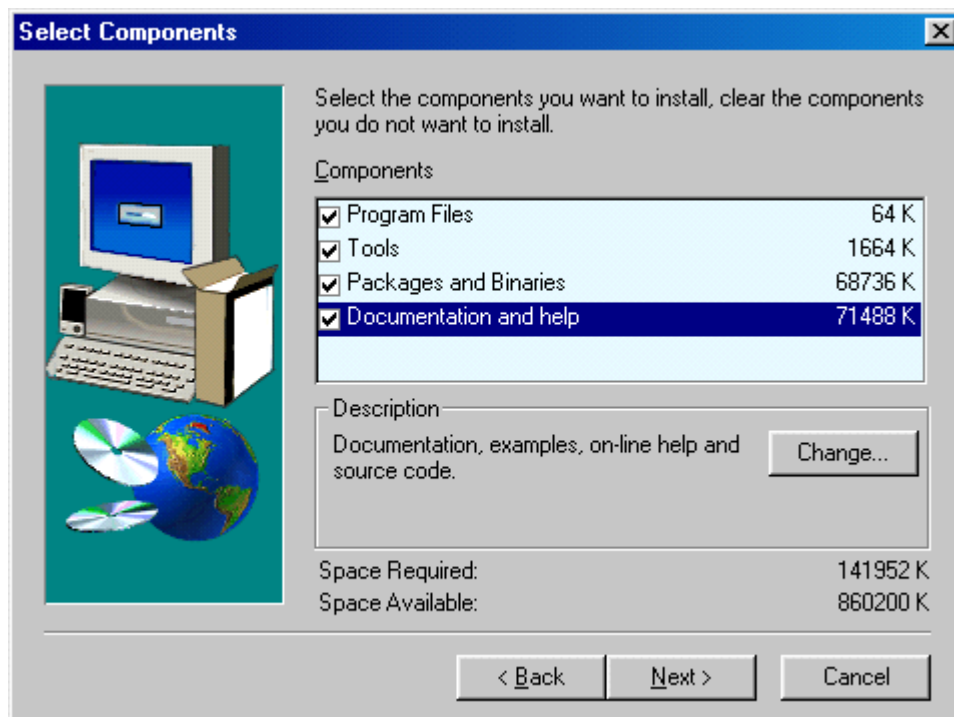


Fig. 2.4. Selecting components for installation

After installation new bookmarks containing visual and not visual components meant for MDA-applications creation will appear on Delphi components palette. The basic components are presented below.

BoldHandles (see Fig. 2.5) contains not visual components for application business-layer generation.



Fig. 2.5. Components palette for business-layer generation

BoldPersistence (see Fig. 2.6) contains not visual components for interoperability with Persistence Layer (DBMS), as it is called in BMDA.



Fig. 2.6. Components for interoperability with DBMS

BoldControls (see Fig. 2.7) contains visual and not visual components for GUI creation.



Fig. 2.7. Components for GUI creation

It is necessary right now to pay attention once more to the fact that visual components of this BMDA version interoperate not with the data, but with objects of business-layer. For this reason BMDA has its own analogues of such visual components, as Label, Grid, etc., possessing additional properties necessary for such interoperation.

ПРИМЕЧАНИЕ

Further it will be shown, that in many cases it is possible if necessary to use usual visual components too, including, components of third-party companies.

BoldMisc (see Fig. 2.8) contains other auxiliary visual and not visual components.



Fig. 2.8. Auxiliary components

Bold COM Handles (see Fig. 2.9) contains not visual components for multilink applications business-layers generation. Such applications allow to distribute functionality among “rungs” so as an opportunity appears to create so-called “thin” clients of databases that are characterized by minimal requirements in respect to presence of means of access to the data. More detailed information will be presented further.



Fig. 2.9. Components for work with multilink applications

Bold COM Controls (see Fig. 2.10) contains visual and not visual components for multilink applications GUI generation.

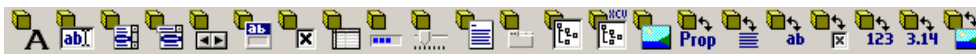


Fig. 2.10. Components for multilink applications GUI creation

Other Bold for Delphi bookmarks will be considered separately, at the description of product additional capabilities.

Summary

In the given chapter the review of Borland Company software products supporting MDA-applications development is resulted.

The history of such products creation totals some years. The information is presented on general characteristics and capabilities of Borland MDA software tools, and also regarding advantages that they give to the developer, reducing time of databases applications development practically by a factor of ten. The base of Borland MDA-technology is the software product created earlier – Bold for Delphi components package, included into Borland Delphi 7 Studio Architect version. At present this Delphi version is the unique Borland software tool allowing to create MDA-applications for Win32 (Windows) platform. The information is given about Bold for Delphi package installation, and also about structure and function of its basic components.

Chapter 3. Quick Start

To familiarize quickly with capabilities of new technology, in this chapter we shall consider how MDA-application is created in practice using Bold for Delphi.

Creating Simple MDA-Application

Creating Business-Layer

Let's create a separate folder for new Delphi-project. We shall create the new project in Delphi, consisting of one form, and we shall save it in the created folder. It will be saved under the name “project1.dpr” by default, and the module will be saved under the name “unit1.pas” by default.

In the panel of Delphi components we shall select <BoldHandles> bookmark. We shall place on the form the following three components from <BoldHandles> bookmark:

- ❑ BoldModel1 (component providing model storage)
- ❑ BoldSystemHandle1 (the basic component-handle of Object Space)
- ❑ BoldSystemTypeInfoHandle1 (the basic component-handle of model types)

These components realize a basis of Object Space of our application. For correct functioning they should be interlinked and adjusted as follows:

For BoldSystemTypeInfoHandle1 component in Object Inspector we shall set BoldModel property in BoldModel1 value (it will appear in the dropdown list). Besides we shall set UseGeneratedCode property in False value (see Fig. 3.1). Such setting for this property means that code generation will not be carried out for model classes of our application.

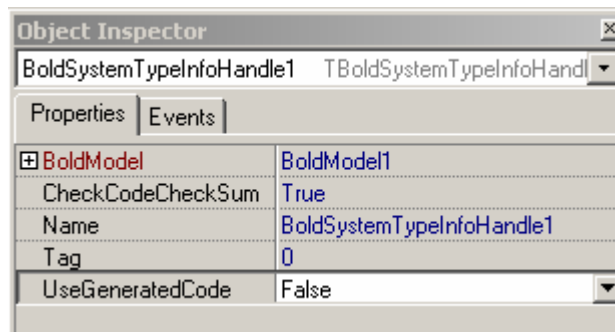


Fig. 3.1. BoldSystemTypeInfoHandle1 setting

For BoldSystemHandle1 component in Object Inspector we shall set SystemTypeInfoHandle property in BoldSystemTypeInfoHandle1 value (it also will appear in the dropdown list). Besides we shall set AutoActivate property in True value (see Fig. 3.2). Thus activation of Object Space “on the first request” is provided.

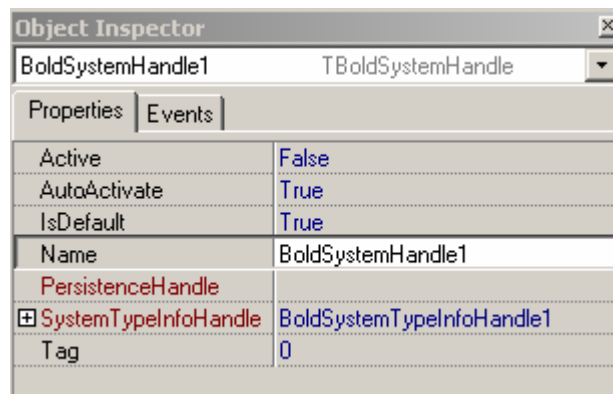


Fig. 3.2. BoldSystemHandle setting

We have created the business-layer prototype for our application. The sequence of the above-stated operations is practically identical for all cases and is always repeated when creating any application using Bold. However business-layer is not filled yet with the functional contents, as by the moment the main thing is not made yet, namely, the application model is not created, according to which MDA-application should function.

Creating Application Model

In our example for model creation we shall use built-in models editor in Bold for Delphi. To jump to the editor we shall click twice on BoldModel1 component located on our form, or click on this component by the mouse right button, in this way we shall cause the pop-up menu, and then turn to its first item: "Open Bold UML Editor". The window of built-in UML editor (see fig. 3.3) will appear (see Fig. 3.3).

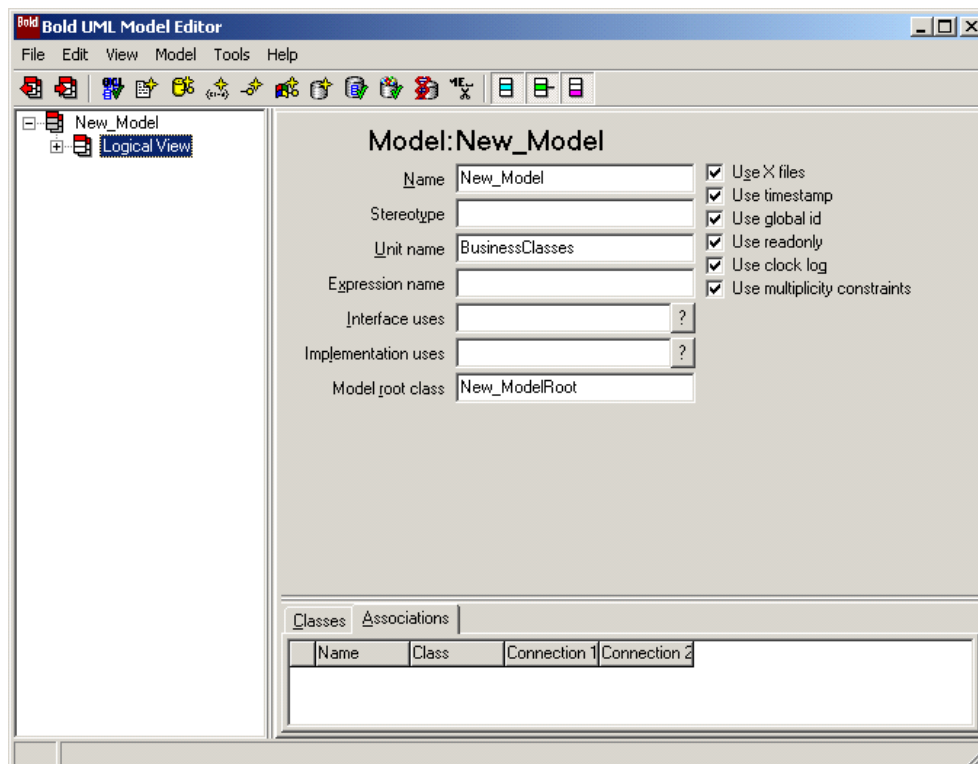


Fig. 3.3. Built-in UML editor interface

This editor allows to create UML-models (in the text mode), and also provides Bold basic functionalities realization at the stage of application development, such, as models export and import from other editors, updating, verification and tweaking of models, databases and code generation. At the present stage we use it for model creation of our simple application.

Before starting direct model creation in UML-editor, we should organize it informally. In this case we shall consider, for example, that the developed application should ensure the functioning with the library catalogue containing the information about authors and books written by them.

For simplification let us assume that the author is characterized by his name and the book by its title.

Besides we shall specially introduce one more artificial condition: each book can have only one author.

Thus, after some further concrete definition, the future application model can be described informally by the following basic propositions (i.e., a set of business-rules):

- ❑ The described application domain includes a quantity of authors and a quantity of books;
- ❑ The author is uniquely identified by text attribute – a name;
- ❑ The book is uniquely identified by text attribute – a title;

- ❑ The author can write many books;
- ❑ The book can be written only by one author;

UML-editor purpose is to transform the informal business-rules description into formal model in UML.

Now we can start model creation.

We shall click by the mouse right button on “LogicalView” item in the top part of the left editor panel and select in the dropdown menu “NewClass” item –new class creation. We shall do this operation one more time, and receive the result presented in Fig. 3.4.

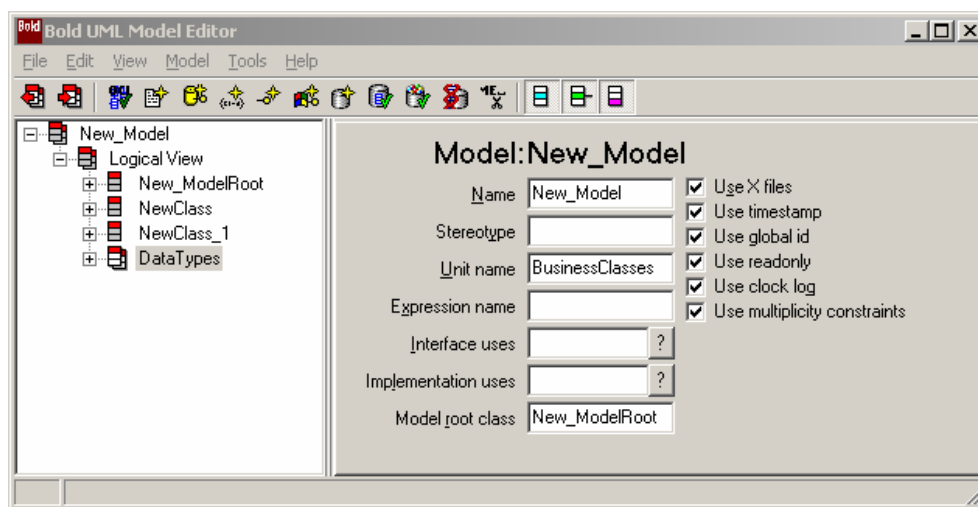


Fig. 3.4. Model view after new classes creation

New classes were named by default “NewClass” and “NewClass_1”. We shall rename them for convenience in the further work. For this purpose in the left panel of the editor we shall select “NewClass” item by the mouse cursor, turn to the right panel of the editor and enter name of our class – “Author” – into Name field, having cleared a former name. For the second class “NewClass_1” we shall set a name “Book” similarly.

After that our model will look like in Fig. 3.5.

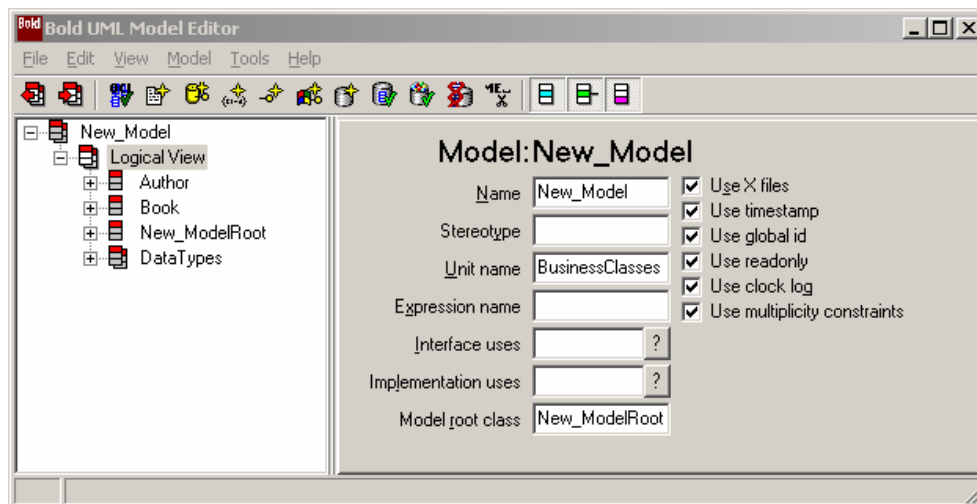


Fig. 3.5. Model view with renamed classes

Now we shall create attributes of our classes. As we have agreed earlier, the author of the book is described in our model by a name of the author, and the book is described by its title. Hence, “Author” class has one attribute, we shall name it "aname", and the “Book” class has also the single attribute, which we shall call "btitle". For attributes creation we shall select the necessary class in the left panel of the editor, click by the mouse right button and select “New Attribute” item from the pop-up menu. We shall create one new attribute for “Author” class and one for “Book” class, and open a model tree in the left panel of the editor, as shown in Fig. 3.6.

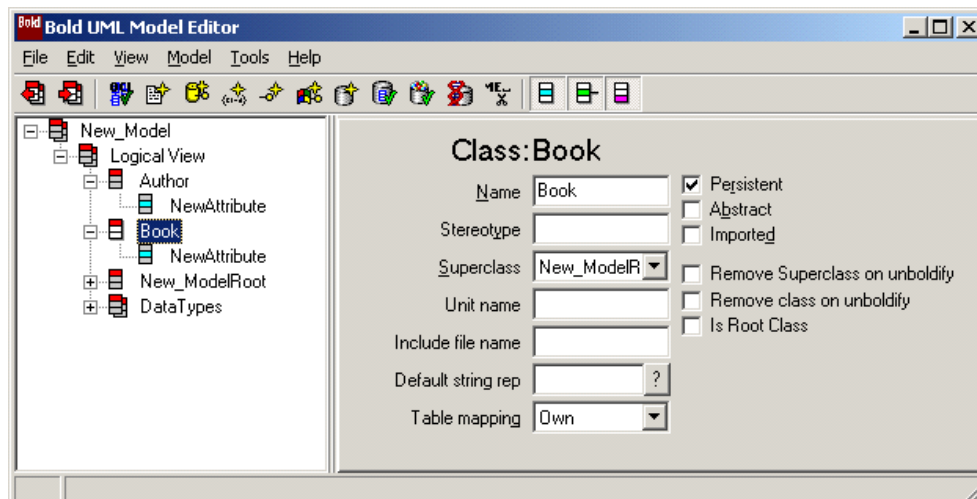


Fig. 3.6. Model view with classes and new attributes

By default new attributes were named (each attribute in its class) “NewAttribute”. Attributes renaming is carried out in the same way as for classes. We shall rename attributes, and get the model shown in Fig. 3.7.

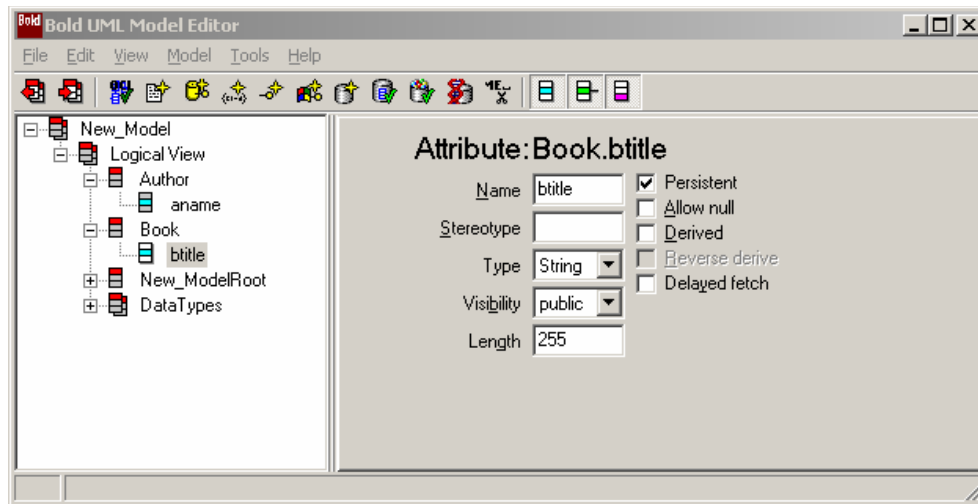


Fig. 3.7. Model view with classes and attributes

Classes and attributes created by us, not for a while yet meet completely the business-rules formulated above. Namely, our model while “does not know anything” about the fact that authors write books, and books are written by authors. That is between classes some link, or relation should appear. For such relation making it is necessary to create the association connecting our classes. For this purpose we act in the same way as at new classes creation: we select “LogicalView” item in the model tree on the left panel of the editor and then push the mouse right button, but now we select “New Association” item from the pop-up menu. After that we shall open the model tree as shown in Fig. 3.8.

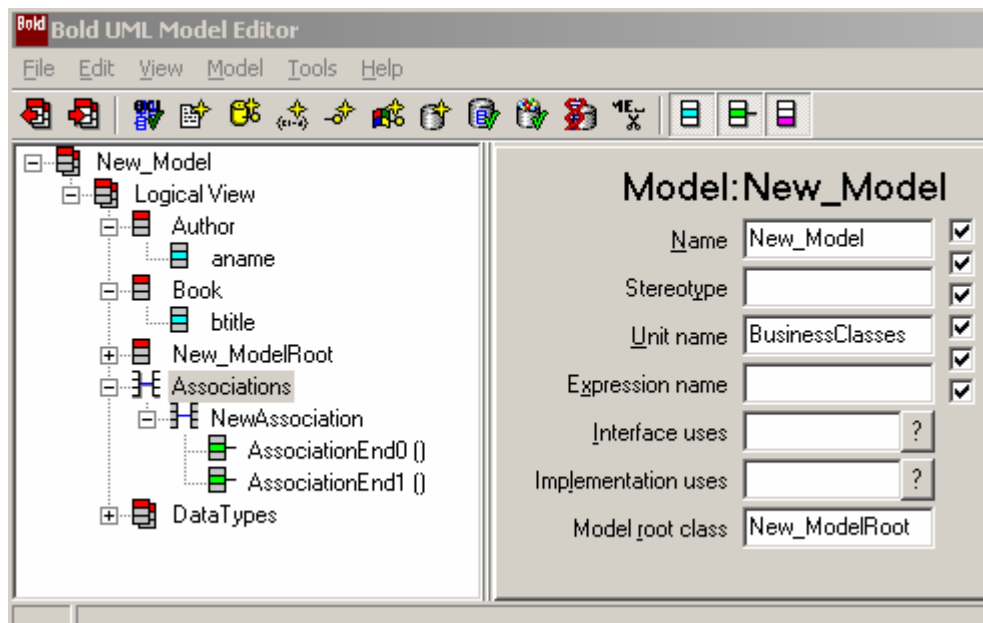


Fig. 3.8. View of the model including association

At the left in the model tree we can see that the created association has two subitems: “AssociationEnd0” and “AssociationEnd1”. These are automatically created names for the ends of our association. The association ends characterize classes roles (see Chapter 1). We shall define the following roles for the ends of our association. We shall select “AssociationEnd0” association role, and in the right panel of the editor we shall write “byAuthor” role name into Name field, and in Class field we shall select “Author” class from the dropdown list; Then we shall set value of “Multiplicity” field as “1..1”. We shall select similarly “AssociationEnd1” association role, and in the right panel of the editor we shall introduce “writes” role into Name field, and in Class field we shall select “Book” class from the dropdown list; we shall set value of “Multiplicity” field as “1..*”. After that the model will look like you can see in Fig. 3.9.

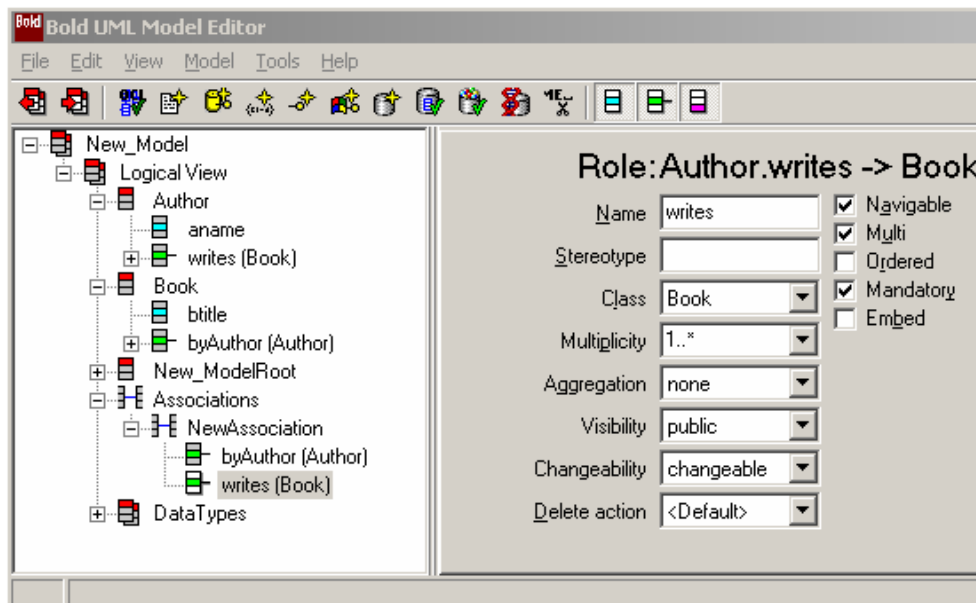


Fig. 3.9. View of the model with association specified roles

Now let's analyze what we have just done with roles of our association and why. As it was said earlier, in our model the simplifying assumption that the book can be written only by one author is accepted. For this reason we have set "Multiplicity" property for "byAuthor" role in "1..1" value, i.e. "one and only one (author)". On the other hand, our model assumes by default, that the author can write many books, and consequently on the other end of our "writes" association "1..*" value is assigned to "Multiplicity" property (see Fig. 3.9), i.e., in other words "one, two.. many (books)". It is necessary to understand now why "1..1" concerns to the author, and "1..*" relates to books. This link was organized by us when appointed classes for the ends of our association, and in Fig. 3.9 this binding, where the window with the name of a corresponding class is located above "Multiplicity" window, is evidently visible.

Creating GUI

So, by the present moment we have created a basis of business-layer, now we have created the application model. The graphic interface is next in turn.

It is necessary to remind at once of one important circumstance mentioned in previous article. When working with Bold it is necessary to get used to business-layer presence in your application which exists not abstractly, but forces you to use special components for access to any structures or data. These components are not visual, and serve as specific translators and implementators of queries which the graphic layer sends to the business-layer for receiving necessary information from the last one. Subsequently we will make sure that business-layer presence provides extraordinary flexibility and convenience. And now we use one of such not visual components for supporting our interface. We shall turn to BoldHandles bookmark and place BoldListHandle1 list handle component on the form.

In Object Inspector we shall appoint BoldSystemHandle1 value to RootHandle property of this component (it can be written manually or selected from the dropdown list).

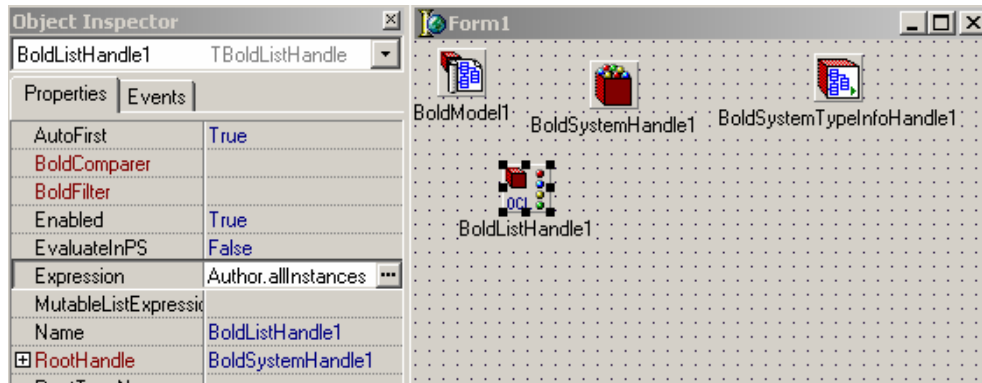


Fig. 3.10. List handle setting

Expression property of the list handle contains OCL-expression text representation (see. Chapter 5). We shall write the following OCL-expression – “Author.allinstances”, as Fig. 3.10 illustrates. That is in our case the list handle will generate “OCL-inquiry” to business-layer which in this case means “I want to receive all instances of Author class”.

Now we shall create directly graphic interface, for this purpose there are quite enough components on a bookmark of BoldControls visual components. We shall select two of them – BoldGrid and BoldNavigator which, as it is easy to understand, are analogues of the corresponding DB-aware components, used when creating in Delphi databases applications in the traditional way. Let’s adjust visual components as follows: we shall set a source of information for both components in Object Inspector– we shall give BoldListHandle1 value to BoldHandle property, and after that we shall click with the mouse right button on BoldGrid1 component, and select “Create Default Columns” item, i.e. “to create columns by default”. In result for BoldGrid1 component “aname” heading of the column (see Fig. 3.11) will be mapped automatically.

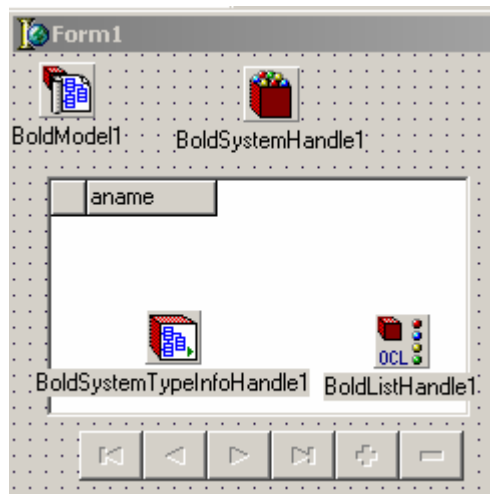


Fig. 3.11. Application form view with adjusted BoldGrid

That's all. We have created our first very simple application, and we can launch it for execution. Using buttons of the navigator, it is possible to add some authors (see Fig. 3.12) to delete or edit them. But, as it is easy to make sure, after exit from the application and its repeated launch all data added manually disappear. It occurs because our application does not contain Persistence Layer yet.

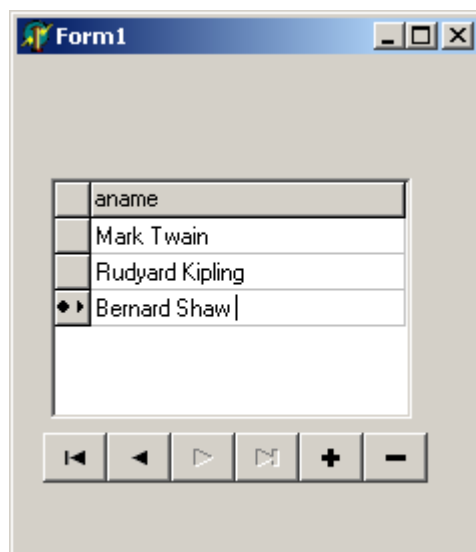


Fig. 3.12. View of application in work

Creating Persistence Layer

So as not to distract the attention now on database creation, pseudonyms adjustment and some other similar things, for persistence layer construction we will use very convenient capability given by Bold for Delphi environment – data storage in XML-document. For this purpose we shall place a component – BoldPersistenceHandleFileXML1 XML-data file handle – from BoldPersistence bookmark on the form.

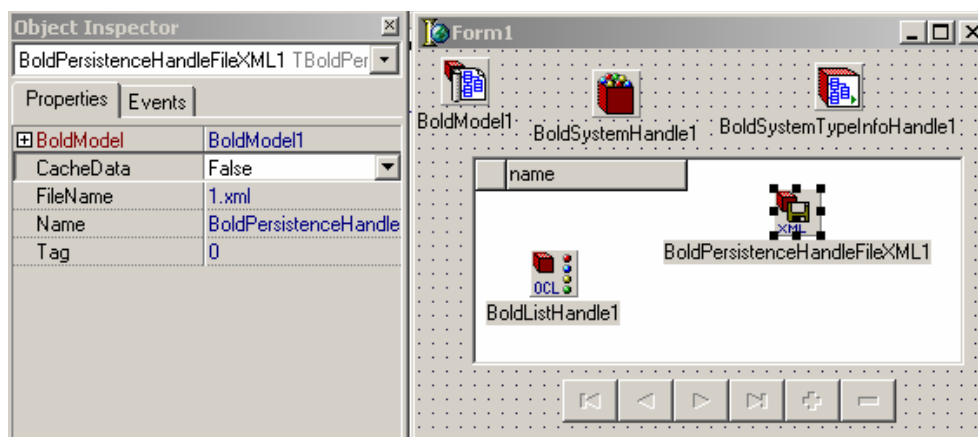


Fig. 3.13. Setting XML-data file handle

For component setting in Object Inspector we shall simply appoint BoldModel1 value to its BoldModel property, and in FileName property we shall write file name, for example, "1.xml", as shown in Fig. 3.13.

Besides for BoldSystemHandle1 component in Object Inspector we shall set BoldPersistenceHandleFileXML1 name of the added component to PersistenceHandle property (see Fig. 3.14).

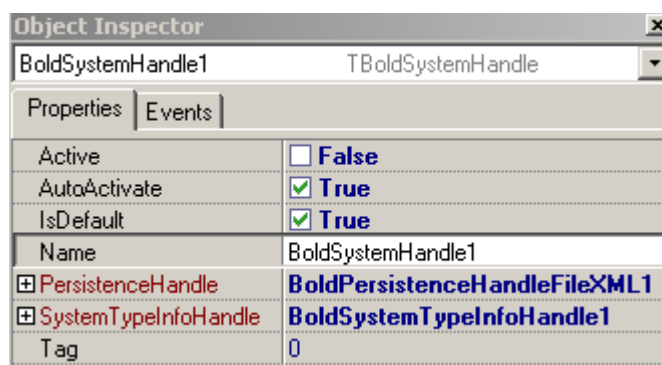


Fig. 3.14. Setting properties binding to the persistence layer

So that our application saves its data, we shall write a code line into procedure of processing “OnClose” event of our form (see Listing 3.1).

Listing 3.1.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    BoldSystemHandle1.UpdateDatabase;
end;
```

Now, as it is easy to make sure, our application has got capability to save the data, and we can study its capabilities more in detail.

Handling Application

At first sight, we have not received anything special yet. We have just created, although not in traditional way, quite usual form for input and editing of the data on authors. By the way, for some reason at that we have not created the form for books titles input - the careful reader can be interested. But the first impression is wrong in this case. To make the closer acquaintance of Borland MDA “magic”, let's add one program module to our application – we shall write in Uses announcement of our Unit1 module the module with BoldAFPDefault name, and then launch the application for execution.

Let's add some authors as shown in Fig. 3.12. Now we shall select any author in a grid and click twice on this line. Thus the new window will automatically open. We did not create this form, for us it was made by Borland MDA environment. The form has some bookmarks, one of which refers to “writes”. Turning to this bookmark, we find out the form for input of books titles. If to proceed to other author not closing the new form, and click twice again on the author name, then one more new form for input of the books written by this author will appear, etc. Thus, we can fill in books cells for all three authors (see Fig. 3.15).

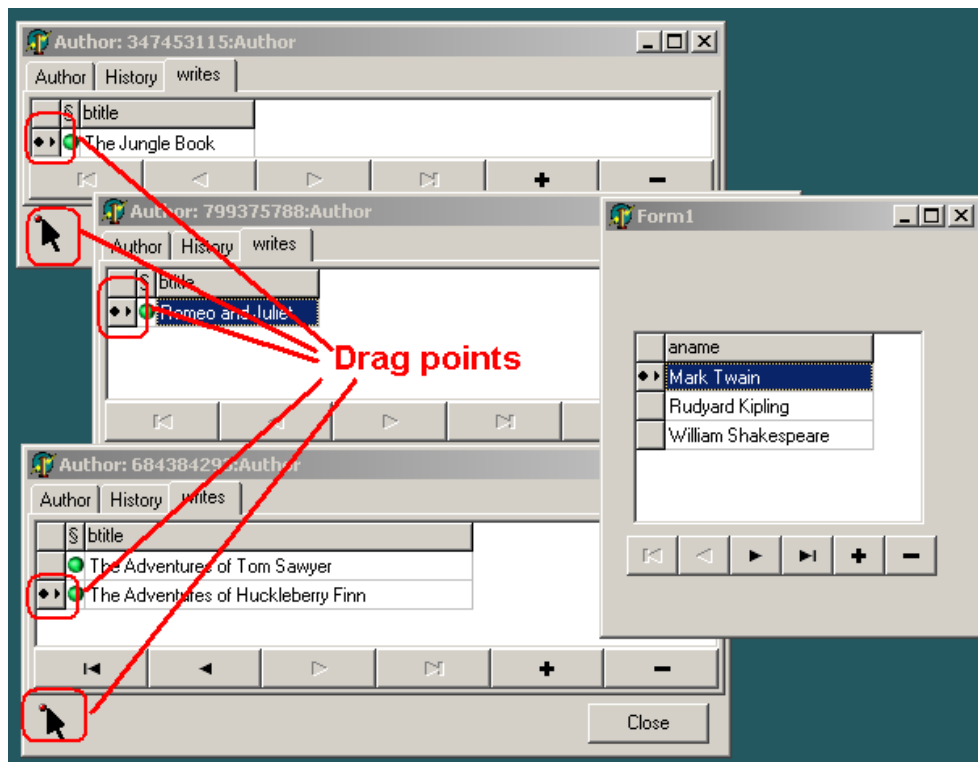


Fig. 3.15. Application with autoforms

It is interesting, that the forms created automatically (autoform) are absolutely independent on cursor position on the main form, and we can edit books of each author absolutely independently. Also it is interesting to make sure, that it is possible “to drag” books from one author to another, if to drag by the mouse not the book title (the second column), but grey cell of the first column on the form of the other author. Also for such “dragging” it is possible to use an area on the autoform marked by the arrow with a point (see Drag points on Fig. 3.15).

The attentive reader will find out without effort, that bookmark “writes” name on the autoform is nothing else but the name of the corresponding association role in our model. And where is the name of the second role in that case? In order to see it we shall select some book on the autoform, and click twice on its title (see Fig. 3.16).

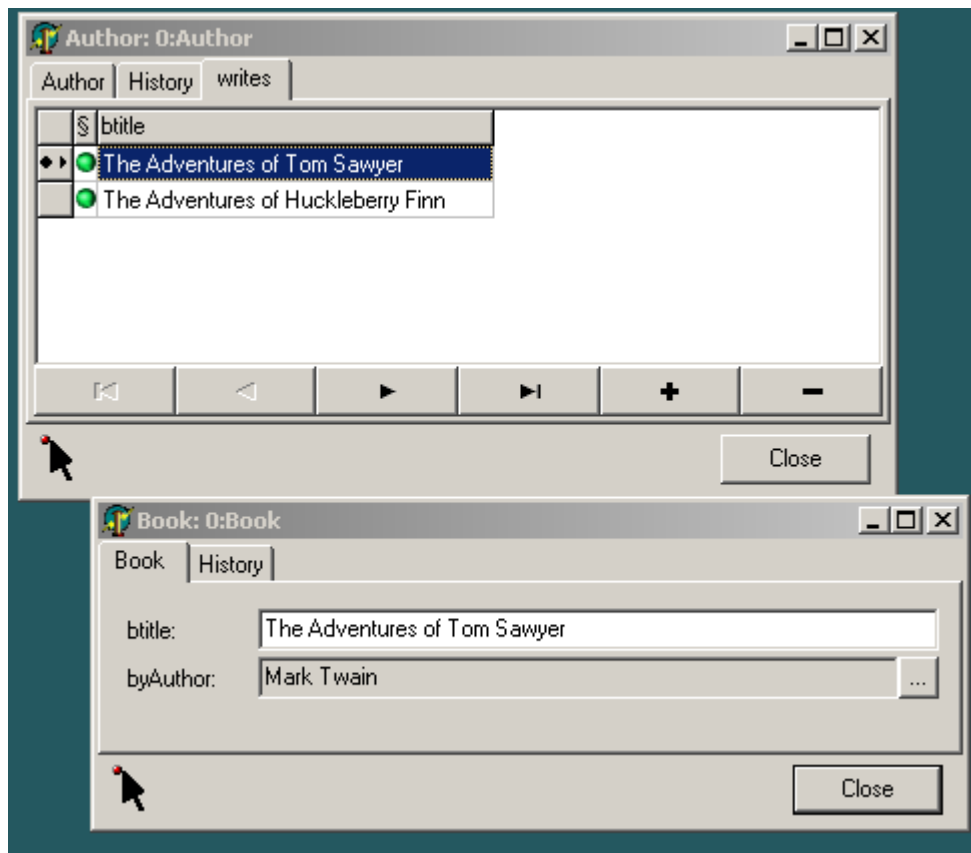


Fig. 3.16. Application with two auto-forms

Thus the new autoform will appear, on which we shall see the book title, a grey field with the author name under it, and “byAuthor” heading to the left of it – this is the second role of our association.

Taking into account, that we practically have not written any code line yet, did not program “drag&drop” mouse events, did not create the form for input of books titles, it is possible to ascertain, that the received application possesses quite good functionality.

In order to see fuller picture taking place, we shall add on the main form BoldGrid2 and BoldNavigator2 visual components from BoldControls bookmark for mapping and managing the books list. For getting the books list of from business-layer we need also the second component – BoldListHandle2 list handle from BoldHandles bookmark. We shall adjust it as follows:

In Object Inspector we shall appoint BoldSystemHandle1 value to RootHandle property of this component (it can be written manually or selected from the dropdown list);

For Expression property we shall enter “Book.allinstances” OCL-expression (see Fig. 3.17).

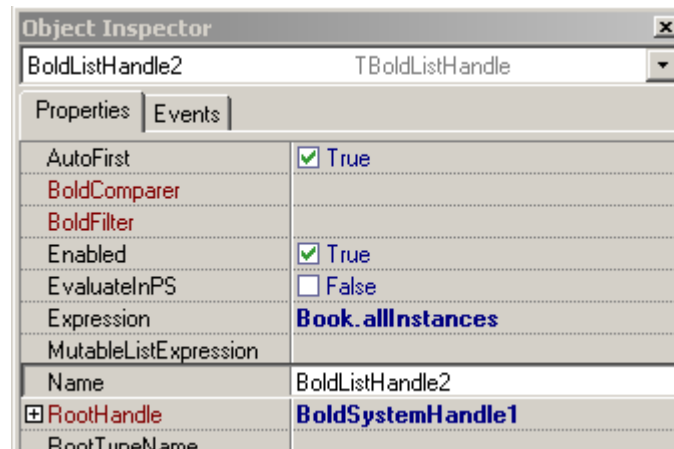


Fig. 3.17. Setting the second list handle

Let's adjust BoldGrid2 and BoldNavigator2 visual components as follows – we shall set a source of the information for both components in Object Inspector – we shall give BoldListHandle2 value to BoldHandle property and after that we shall click with the right button of the mouse on BoldGrid2 component, and select “Create Default Columns” item, i.e. to create columns by default. In result “btitle” column heading will appear for BoldGrid2 component (see Fig. 3.18).

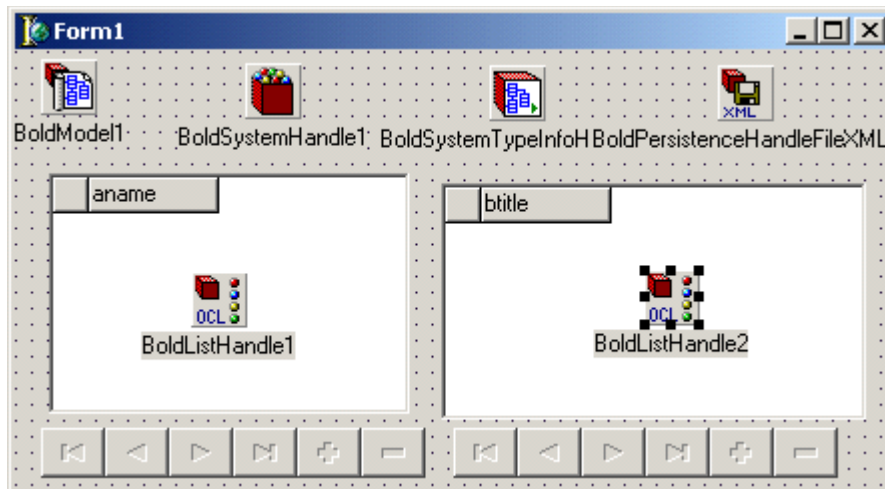


Fig. 3.18. The form view with two grids of data mapping

You can see that Bold automatically selects “aname” and “btitle” columns names in BoldGrid1 and BoldGrid2 grids from the attributes assigned by us in “Author” and “Book” classes. It occurs when we create columns by default. But, naturally, it does not mean, that we cannot adjust BoldGrid tables in other way (for example to give them national language names, see Chapter 9), we just choose the most simple and fast way now.

Let's run our advanced application for execution. We shall add two authors: "Karl Marx" and "Friedrich Engels", with the help of the navigator. With the help of the second navigator we shall add "Dutch Ideology " book. We shall click twice on the name of this book (see Fig. 3.19).

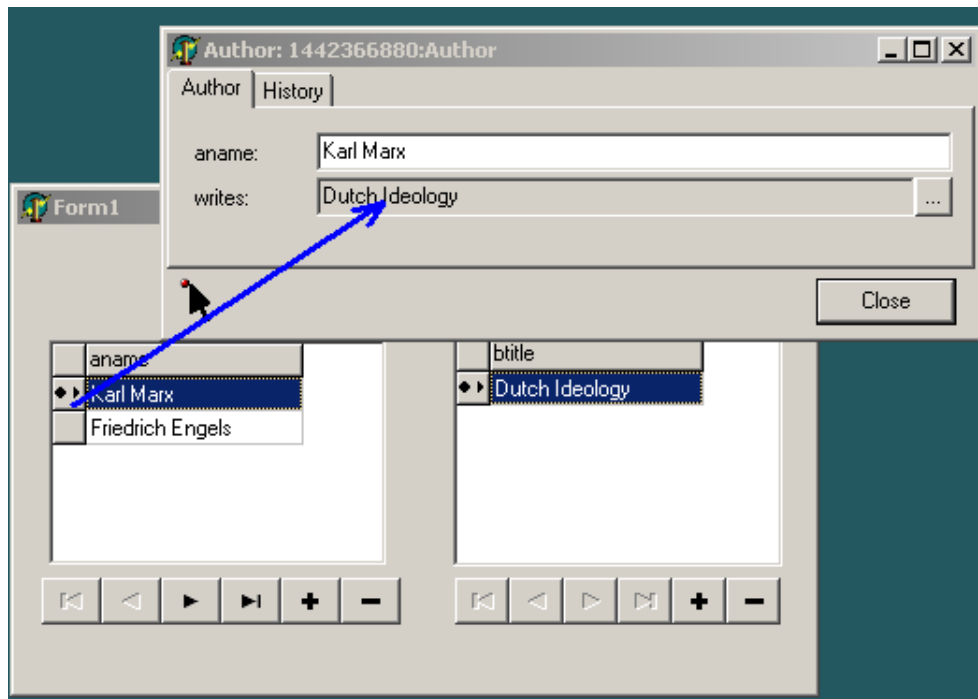


Fig. 3.19. Operating application with two grids and autoform

Let's drag "Karl Marx " author from a grid on the autoform onto the grey field with a name of the author (an arrow in Fig. 3.19). We shall see, that "Karl Marx" author was assigned to "Dutch Ideology " book. We shall drag "Friedrich Engels " author on the same place, and we shall see that "Friedrich Engels " became the author of the book, and "Karl Marx " has disappeared. We understand that the given book actually has two authors: "Karl Marx " and "Friedrich Engels ". But when creating the model we introduced a business-rule: each book can have only one author, and our application, functioning within the framework of the specified model, "does not allow" us to add the second author.

Updating the Model

Let's look, what occurs if we remove the constraint regarding the number of the book authors. We shall formulate the following business-rule instead of the previous one: "each book can have one or several authors". For this purpose we shall launch the models editor, select "byAuthor" role and set "1..*" value for its "Multiplicity" property (multiplicity of relation) (see Fig. 3.20).

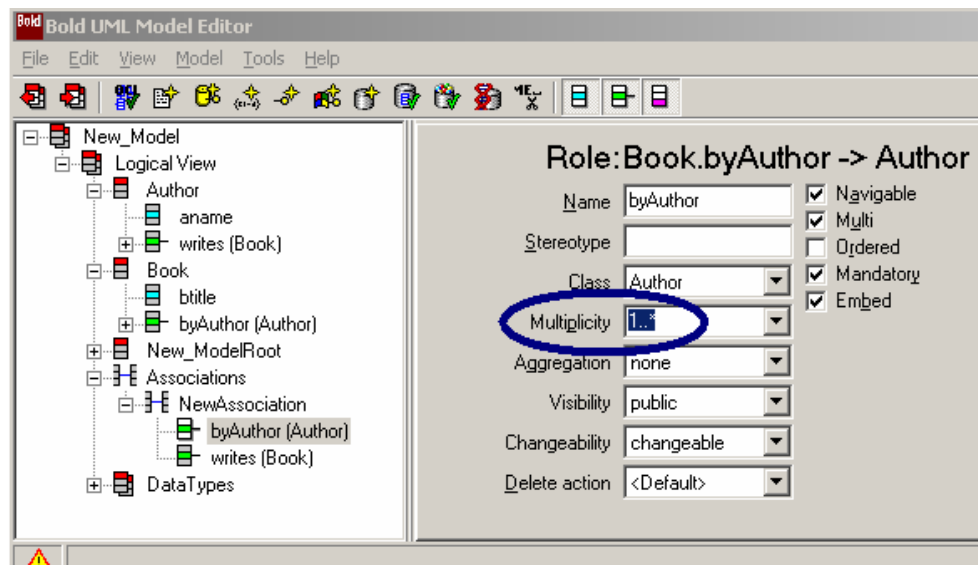


Fig. 3.20. Changing the role multiplicity in UML-editor

Let's run the application. And ... we shall receive the error message (see Fig. 3.21).

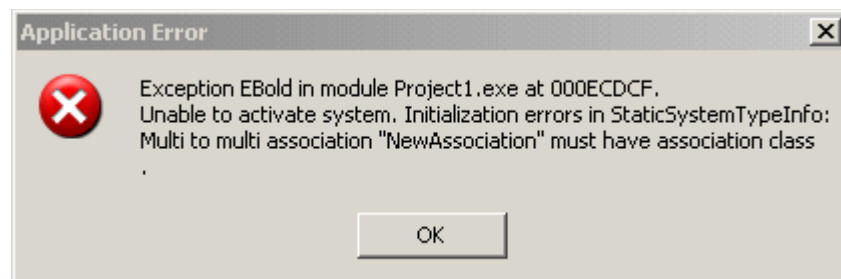


Fig. 3.21. Run-time message about class-association presence necessity

If we try to save our form with the changed model before the application launching, we also shall receive the message (see Fig. 3.22).

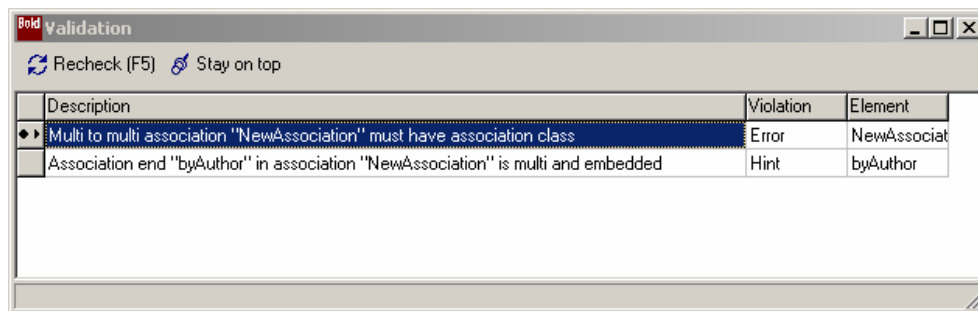


Fig. 3.22. Design-time message about class-association presence necessity

Both messages tell about necessity of class-association presence in the model, providing storage of the information on “many - to – many” link. Really, if to draw an analogy with relational databases, then for cases of such links it is necessary to create the additional binding table. In our case Bold for Delphi environment allows to create such class automatically. For this purpose in the main menu of the built-in UML-editor we shall select “Tools ► UnBoldify Model” item, and right after it – “Tools ► Boldify Model” item. In result in the model tree the new class (see Fig. 3.23) will appear with NewAssociation name.

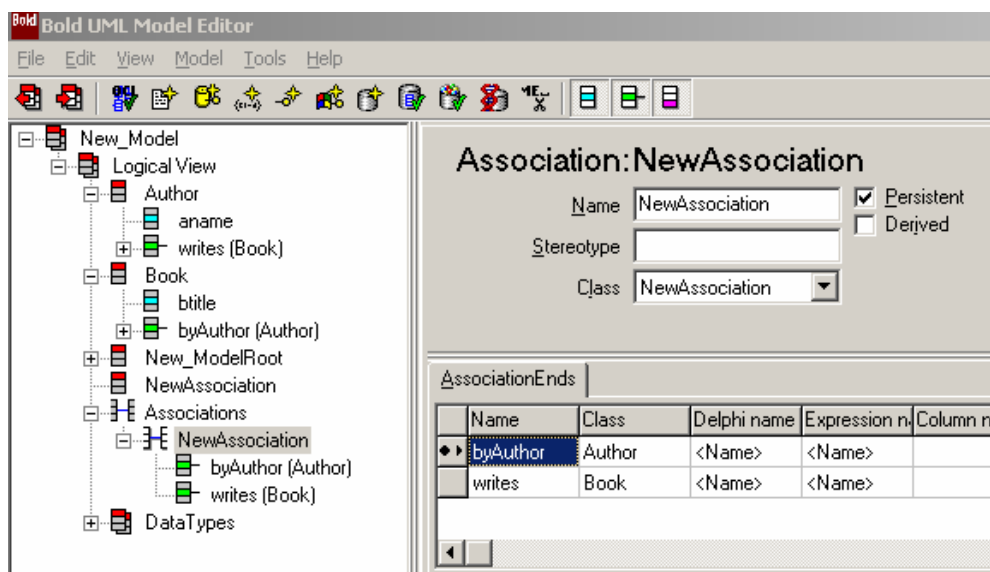


Fig. 3.23. Model view with class-association created automatically

Thus, if we select association in the model tree, then on the right in Class window we shall see a name of the new automatically created class (Fig. 3.23).

Let's save our project, remove “1.xml” file, and launch the application for execution again. We shall add the same two authors with the help of the first navigator. With the help of the second navigator we shall add “12 chairs” book.

Let's click twice on the name of the book. And we shall see, that the autoform has changed (see Fig. 3.24).

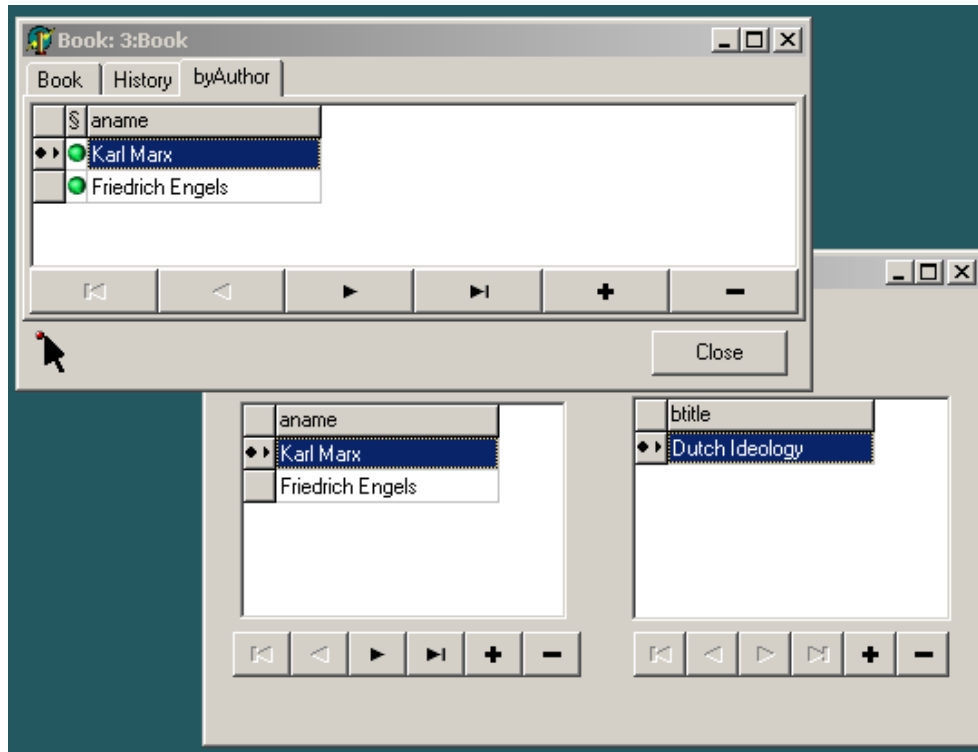


Fig. 3.24. New view of autoform for entering several authors

Now, instead of one window for the author name we find out the form, which allows assigning any set of authors to the given book. We shall drag consistently both authors on the autoform, and receive (see Fig. 3.24), that now both authors are available for “Dutch Ideology” book.

Just now we have observed in practice demonstration of characteristic and basic feature of MDA-applications, namely, MDA-application behaviour is determined not by an application code, but by UML-model. In fact, we have not corrected any code line (strictly speaking there is no code, at least there is no any code written by us), but we have received new functionality of our program. Why? Because we have changed the application model, and it turned out, that it is quite enough for changing its behaviour.

Let's continue studying properties of the created application. We shall add new author: “Dreiser”. We shall add books into the second table: “American Tragedy”, “Genius”, and “Financier”. We shall click twice on “Dreiser” author, open “writes” bookmark on the appeared autoform of the author, then we shall drag these books from the main form onto the autoform of the author (see Fig. 3.25). We shall remind once again, that at dragging it is necessary to place the cursor not on the book title or the author name, but on the cell of the first column marked by rhomb and arrow symbols. These symbols show the current position of the pointer.

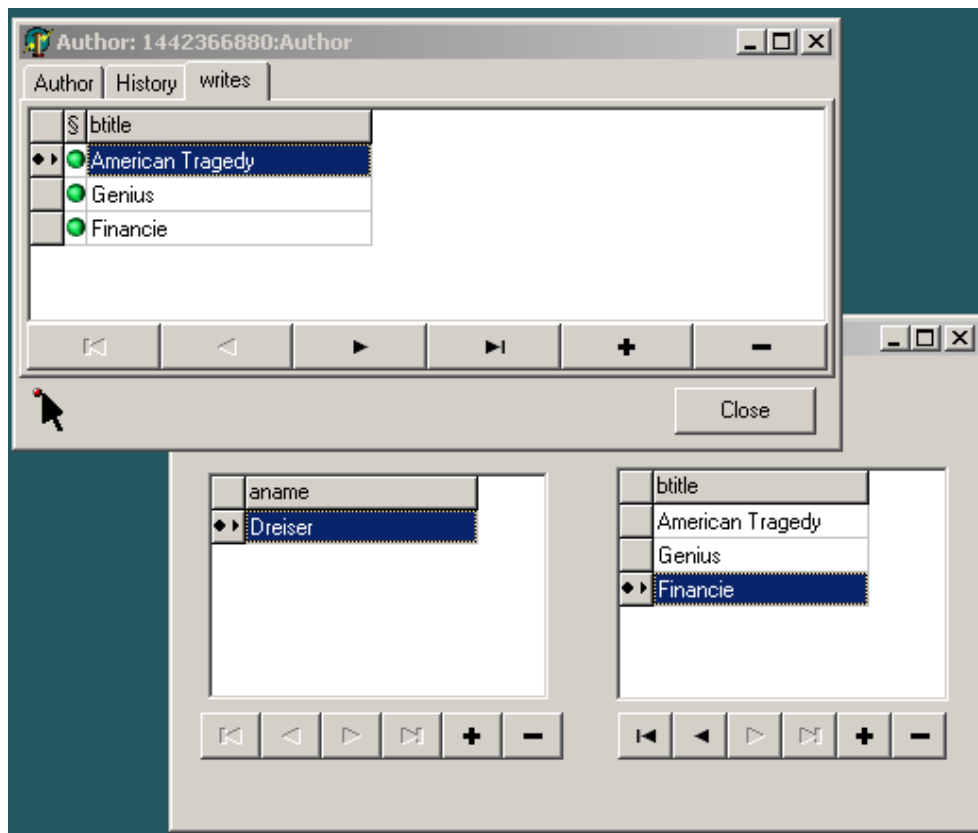


Fig. 3.25. Adding several authors on autoform

And now, when we have assigned these books to “Dreiser” author, we shall delete this author with the help of the first navigator. Thus there will be a window with the requirement of deletion operation confirmation, and after confirmation “Dreiser” author will disappear from the list of authors; at that the author autoform will be closed automatically. But books of this author will appear in the list of books on the main form as before. Such behaviour of the application is not always acceptable. There is no author, and his books are present. That is some “data corruption”, as the saying goes when working with BMS.

Let's try to change situation. We shall formulate changed earlier business-rule a little otherwise: “each book should be written at least by one author”. Thus, we say, that there should not be books which do not have author.

For realization of this rule we will address again to our model, we shall select “writes” role of association and change “Delete Action” parameter value from <Default> onto “Cascade” (see Fig. 3.26).

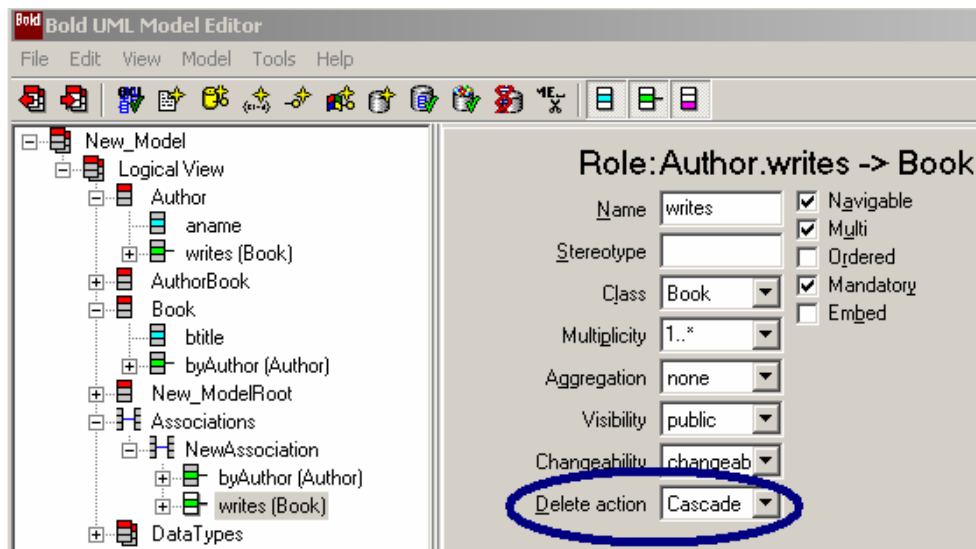


Fig. 3.26. Changing UML-model properties

We will launch our application again. Let's add "Dreiser" author whom we have deleted last time, we shall assign to him three books – "American tragedy", "Genius", and "Financier" by way already known.

Now we shall try to delete the author from the main form again. And we make sure, that after his deletion from the right list of books, the books written by him also have been deleted. We have seen again, how change in the application model affects directly its behaviour.

Discussing Results

Let's estimate functionality of the created application, from new position already. We have constructed, from the point of view of applied capabilities, the classical application for work with a local database which contains such links as "many - to - many". At that:

- ❑ The database was not created actually, but UML – model was created;
- ❑ Tables and fields, primary both secondary keys and indexes were not created, but the classes containing attributes and associations with roles were created;
- ❑ Such links as "wizard – slave" and "many – to – many" were not created, but dimensions of roles (multiplicities of relations) were set;
- ❑ Intermediate binding tables for realizing "many – to – many" relations were not created;
- ❑ SQL was not used;
- ❑ Forms for data entry and editing were not programmed, they have been created automatically;
- ❑ "Drag&drop" interface was not programmed.

The readers familiar with development of databases applications in Delphi, probably can estimate really time spending for creation of the similar application by traditional way. And also they are able to estimate effort at modification of database structure, that we made, as a matter of fact, when changing links dimensions for authors in our model.

In addition we can say that even such simple application is flexible sufficiently, at that it does not require, as we have seen, a program code writing . Our application does not generally contain a program code (we have in view the user program code certainly).

Also it is necessary to note that we have gone not in the most optimum way when used Bold built-in models editor. Unfortunately, it is not graphic. As we will see further, using graphic UML-editors can lighten essentially and even more speed up work on creation and updating of applications model.

Creating Standard Applications

In the example mentioned above we created the application “manually”. However, after Bold for Delphi package installation, there is an opportunity of some automation of this process. If in the main menu of integrated Delphi environment we select consistently File ► New ► Other ► BoldSoft items, and select in the drop window the item (single) named “Default Bold Application” (see Fig. 3.27), then the template for the new Bold-application will be generated automatically.

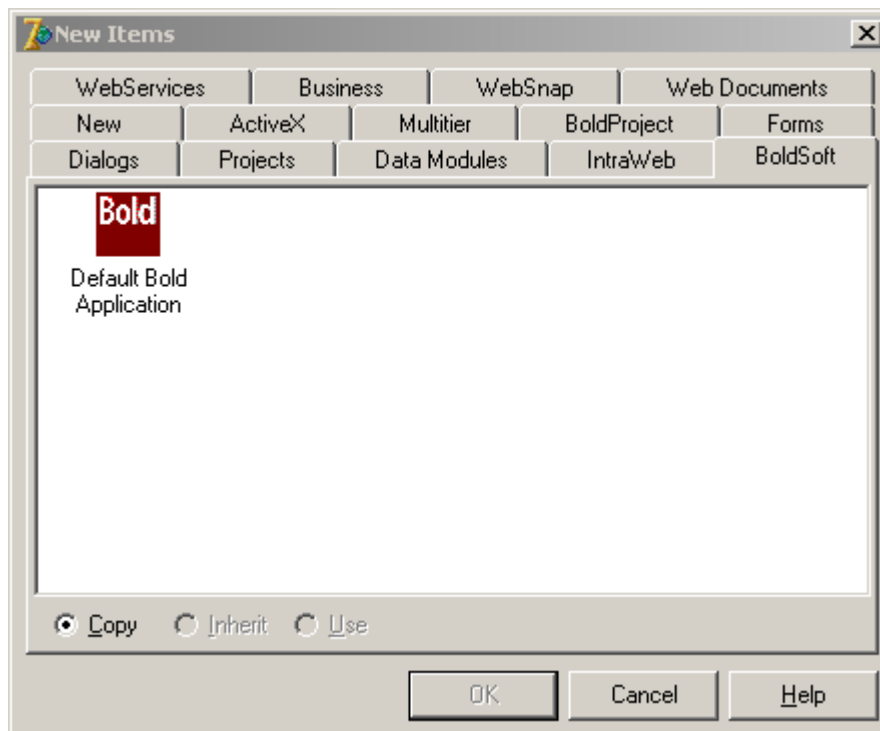


Fig. 3.27. Creating standard application

This template includes three forms:

- The module of dmMain data; includes a set of Bold-components for object space creation, a component for link with RationalRose, DBMS Interbase adapter, and also ActionList1 and IBDatabase1 standard Delphi-components (see Fig. 3.28);

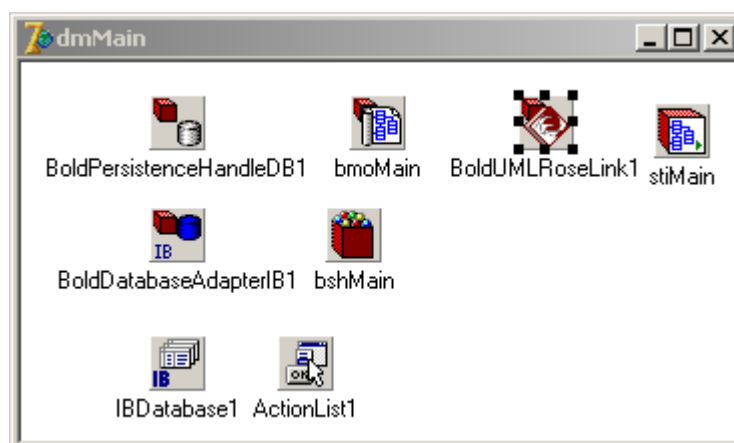


Fig. 3.28. Data module created automatically

- ❑ FrmMain main form; the form contains the main menu, and also a text field with instructions (in English) on the further actions of the developer (see Fig. 3.29);

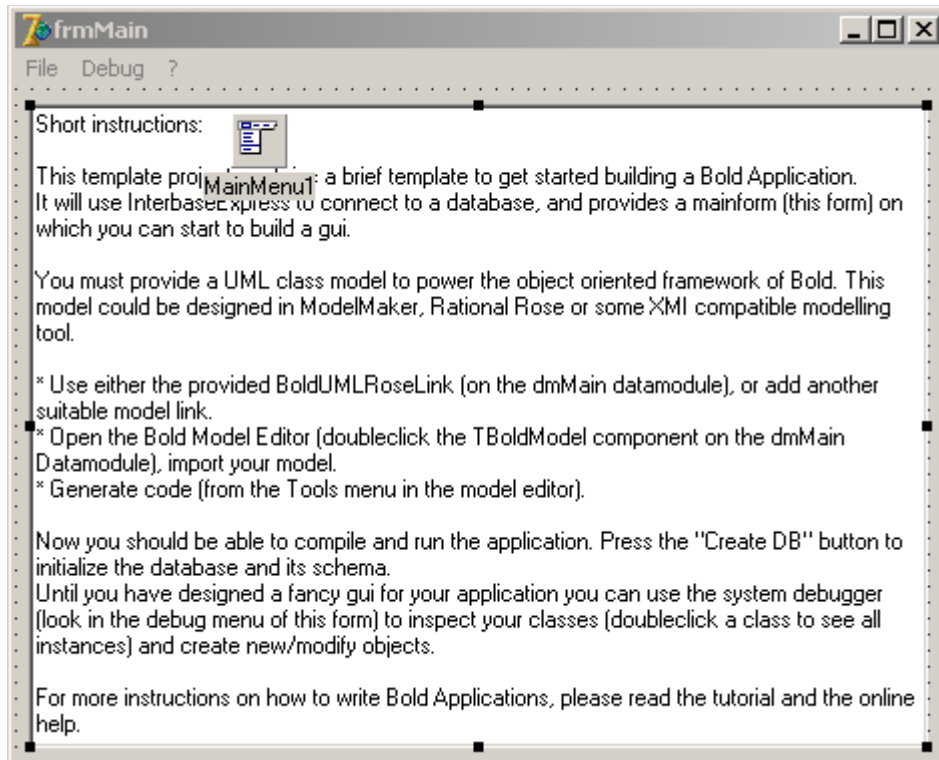


Fig. 3.29. The main form

- ❑ FrmStart “starting” form, which contains buttons for database creation, object space activation and cancellation (see. Fig. 3.30)

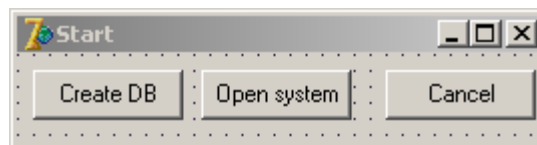


Fig. 3.30. The starting form

Implementation of the mentioned way of creating applications standard “preliminary models” is rather convenient, however, it requires additional knowledge for the further components setup and the application development. During the further studying all necessary information will be given in this book. Now we shall only note, that it is possible if necessary to use a way of separate addition of such standard “preliminary models”, i.e. not the application as a whole, but the concrete form or data module. For this purpose it is

necessary to select consistently File ► New ► Other ► BoldProject items from Delphi menu, and to select in the drop window a desirable component, which is required to be added into the project (see Fig. 3.31).

NOTE

“BoldProject” bookmark becomes accessible for separate addition of standard forms only after creating standard “preliminary models” of the new Bold-application. That is, if the “usual” project in Delphi has been started, this bookmark will not be mapped and it will be impossible to use it.

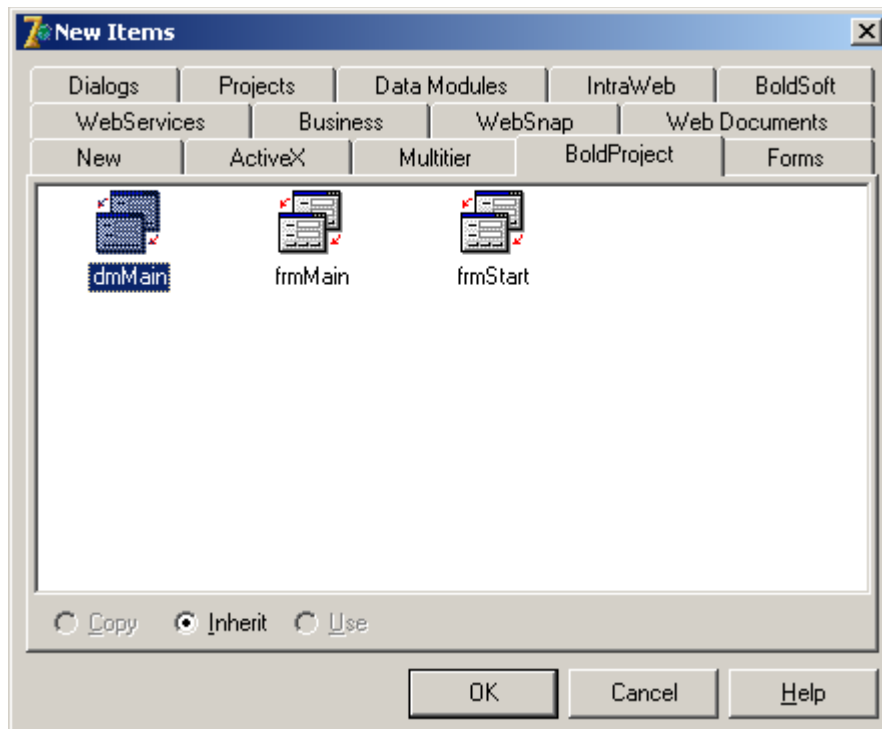


Fig. 3.31. Addition of different forms into Bold-project

Summary

By the example of the simple application MDA software tools use is shown.

- ❑ It is shown in practice that MDA-application is organized as a “three-layer pie”, and even most elementary application contains three layers: persistence layer, business-layer, and an applied layer (GUI).
- ❑ It is demonstrated that the graphic interface addresses to the data through the intermediary – business-layer, for this purpose there are special not visual components.

- ❑ It is shown that MDA-applications functioning is impossible without model creation. And, on the other hand, the application behaviour is determined by this model not abstractly, but absolutely specifically.
- ❑ Even on such simple application advantages of MDA-technologies use are shown. Extrapolating the considered simple example on the corporate information systems containing hundreds of classes and thousands of attributes, it is possible to present approximately just now, why the described technology allows to increase cardinally efficiency of their development and support.

Chapter 4. Application Model

As it was said earlier, the application model is a basis of MDA-technology. The model contains composition, structure and elements of the developed application behaviour. In this chapter we shall consider practical work with UML-models, and also we shall learn to use Rational Rose CASE-system jointly with Bold for Delphi software toolkit.

Model Role in Borland MDA

From Bold for Delphi point of view the model has the following functions:

- ❑ Determines composition and type of classes and classes attributes;
- ❑ Fixes occurrence of links between classes, defines type of these links and their dimension;
- ❑ Specifies conditions and constraints, imposed on classes and their attributes; such conditions are formed in Bold built-in OCL dialect;
- ❑ Contains the information for adaptation to Delphi development environment as a set of special *tagged values*;
- ❑ Contains the information for adaptation to DBMS, also as a set of special tagged values.

Thus, in Bold there is no obvious clear-cut distinction between MDA PIM- and PSM-models (see Chapter 1), and one common model contains all necessary information. PSM-model functions are basically carried out by means of tagged values, which will be described further in this chapter. Besides some Bold components being DBMS adapters are able to carry out such functions too (see Chapter 10).

Bold for Delphi uses model for realizing the following basic functions:

Code Generation. Code Generation means automatic creation of descriptions modules (.inc - files) and functional modules (.pas - files) in Object Pascal language, containing full realization of application model classes (for more details see Chapter 12). Strictly speaking, Bold in itself does not require code generation. As we have already seen by the example of the simple application (see Chapter 3) and shall see further, it is possible to create grave enough applications, not using code generation at all. However in some cases code generation proves to be necessary. General strategy looks here approximately as follows: if it is not enough OCL capabilities supported by Bold for realizing, for example, desirable methods or calculated attributes at business-layer, the developer has the right “to go down” to the level of a program code. Besides, code generation is necessary, if model classes contain *operations*. Code generation can be carried out from the model built-in editor (Model Editor) at the application development stage. In addition to code generation, Bold also provides an ability of interfaces generation necessary for functioning of distributed DCOM-applications.

Database Scheme Generation. The database structure (i.e. a set of tables, fields, indexes and keys descriptions) can be generated automatically both from the built-in model editor at the development stage, and by means of software tools during the application execution.

Bold also supports synchronization of database structure with the application model changing in time (*Model Evolution*), thus information already available in DB is saved. In Bold such capability is named *DataBase Evolution*. It is necessary to keep in mind, that there is no one-to-one correspondence between model classes composition and structure of generated database tables. First, Bold uses a number of DB tables for its own needs and creates them automatically. Second, not all model classes require saving the objects information in DB (persistent class), and can be represented as transient classes, besides, even in model persistent-classes the derived attributes are frequently present, which are not saved in database. And third, in some cases it is essentially impossible to realize some kinds of relations in relational DB. So, if it is quite allowable in UML-model to link two classes by the association having multiplicities more than 1 on both ends, then in relational database realization of such kind relations (“many-to-many”) needs creation of the intermediate binding table. Therefore when generating the database scheme in such cases Bold is able to generate such tables independently and insensibly for user.

Interpretation of model during execution for managing the application behaviour. The information on model is kept in TBoldModel component, and at the stage of the application run it is used for object space management, for its integrity control, and also for managing interoperability of business-layer with persistence layer and GUI. One can say that **the Object Space in Borland MDA is an instance of model**, much as the object is an instance of class in OOP. Management of each object space is provided with TBoldSystemHandle component. This component receives the information on the types contained in model from TBoldSystemTypeInfoHandle component.

Tagged Values

Besides UML-model BMDA for the functioning uses a set of special variables-parameters, or tagged values. They are necessary for interoperability with development environment and DBMS, and also for model additional settings.

Tagged values appearance is not randomness. If the UML-model can be considered as platform-independent PIM-model (see Chapter 1), then set of tagged values can be considered as platform-dependent PSM-model. Being linked with PIM by this reason, tagged values are classified by ownership to elements of model hierarchical structure. Thus, there are separate sets of tagged values for the following elements of model hierarchy:

- ☐ Model as a whole;
- ☐ Class;
- ☐ Association;
- ☐ Attribute;
- ☐ Role;
- ☐ Operation.

On the other hand, tagged values can be classified according to functional ownership too. Tagged values are subdivided on the following groups:

- ☐ Common;
- ☐ Used for generation of code and interfaces;

- ❑ Used for Persistence Mapping;
- ❑ Used for description of the ownership to persistence layer.

The developer also can create his own tagged values and use them for his purposes. Tagged values are accessible both at the application development stage, and during its run. Borland MDA described version includes several dozens of tagged values. We shall familiarize with them more in detail at the description of technology of applications models development.

Rational Rose as Means of Models Development for Borland MDA

In the previous section the basic functions of the application model in Borland MDA have been considered, and also UML classes diagram elements are described briefly. The present chapter is dedicated to practical development of MDA-application model in graphic UML-editors by the example of Rational Rose CASE-system. It is necessary to note, that, after inclusion in Bold for Delphi version 4 complete support of model import and export functions in XMI format (XML Metadata Interchange), the capability to develop applications model for Borland MDA in any UML-editor supporting this format (for example, in PowerBuilder of PowerSoft Company) basically has appeared. Besides Delphi 7 Studio structure contains ModelMaker tool also including advanced means of UML-modelling and some tools for integration with Bold. Nevertheless, with the big share of confidence it is possible to assert that for the present moment Rational Rose remains the most convenient means of developing UML-models for Borland MDA. The matter is that as the tool of creation of model for Bold, Rational Rose CASE-system takes a special place among other software tools possessing the graphic UML-editor. Interoperation with Rational Rose is incorporated soundly in Borland MDA since early versions of Bold for Delphi product. This interoperation is realized by means of COM technology (Component Object Model). COM detailed description is given in many sources, for example, in [3].

From COM point of view Rational Rose is *the automation server* carrying out requests of the client (*controller*) – Borland MDA development environment. Due to such close interoperability mechanism, the following useful functionalities are provided:

- ❑ Unattended startup of Rational Rose by query from Delphi environment;
- ❑ UML-models and tagged values import from Rational Rose into Bold;
- ❑ UML-models and tagged values export from Bold into Rational Rose;
- ❑ Access to Bold tagged values when developing model in Rational Rose;
- ❑ Rational Rose adaptation to the specific Bold versions.

The functions just listed allow to unite in practice convenient expressive means of Rational Rose graphic interface with capability of tweaking the application model in Borland MDA environment.

Rational Rose Setup

Before model creating in Rational Rose editor it is necessary to provide its presetting for work with Bold for Delphi. For the beginning it is necessary to make certain that Bold for Delphi is available in “Add-In” of Rational Rose. For this purpose it is necessary to call Add-In Manager window through main menu ► Add-Ins ► Add-In Manager ... (see Fig. 4.1), and then to activate Bold for Delphi item, in case of need.

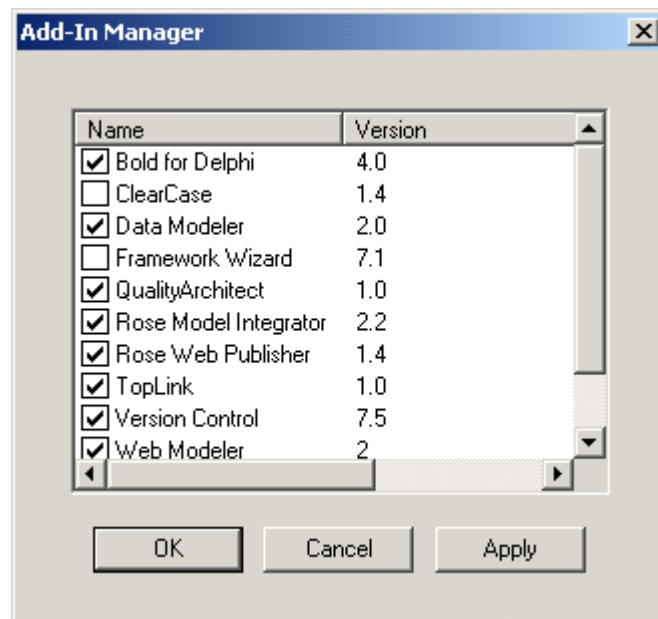


Fig. 4.1. Rational Rose Add-In Manager setup window

For adjustment under concrete Bold for Delphi version the special file of properties (Properties Files) with the name "BfD.pty" is used; it is created at Bold for Delphi installation in <... \Program Files\Boldsoft\BfDR40D7Arch\Rose \> folder.

For its use it is necessary to select in Rational Rose Tools- ► Model Properties ► Update ... menu items, and then select the above-stated file. After that it is necessary to proceed to the window of model settings (Tools ► Model Properties ► Edit), select "Bold" bookmark, then select from the dropdown list "Project" parameter, and set new value to its "PTYVersion" property (the lowermost in the list of parameters), see Fig. 4.2, where 6.4 number of model version corresponds to the last product updating.

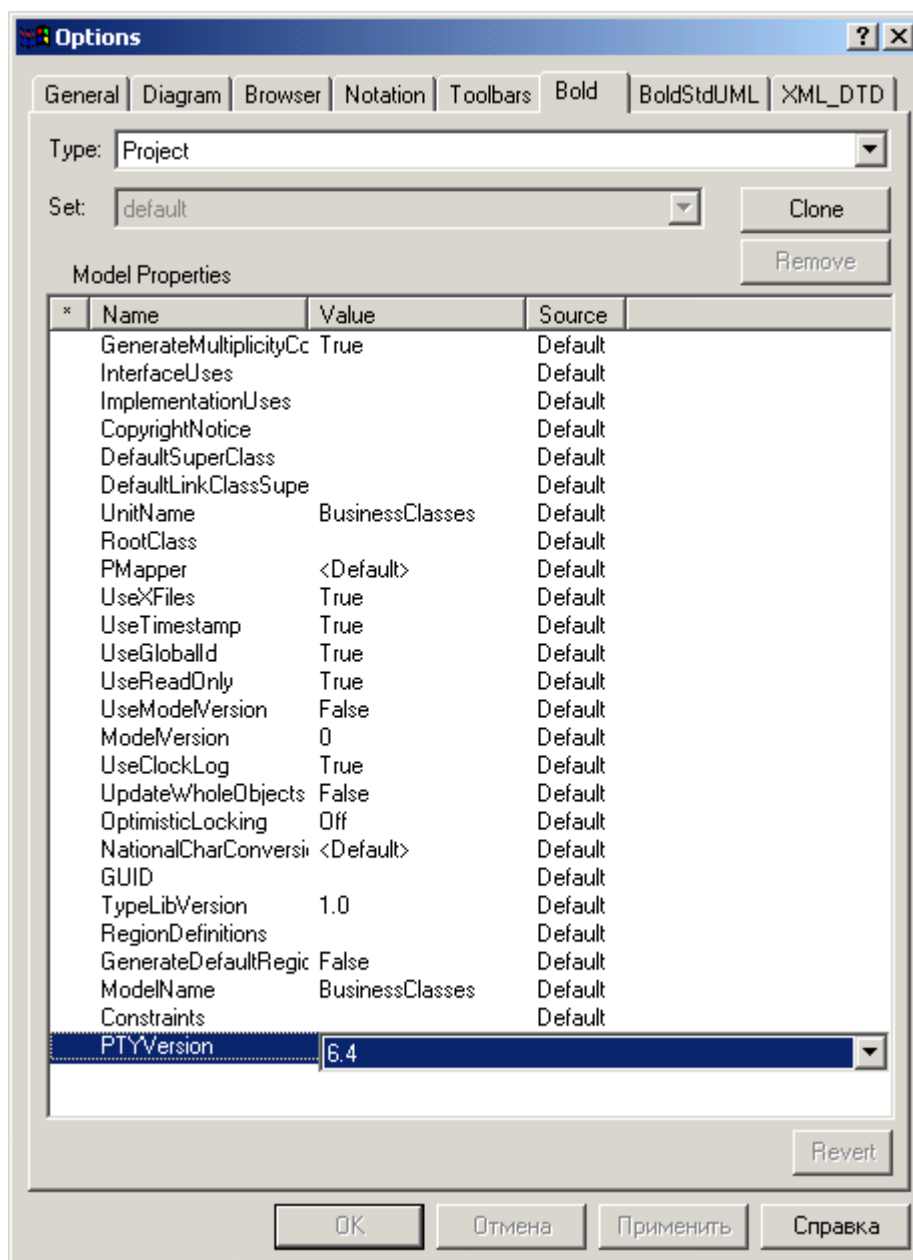


Fig. 4.2. Rational Rose model setting for work with BMDA concrete version

Here adjustment stage of Rational Rose is finished. At each Bold for Delphi version updating this Options process should be repeated

Creating Model in Rational Rose and its Import into Borland MDA

In the previous chapter application model creation the by means of Bold built-in text UML-editor has been shown. By the example of the similar model we shall look, how is it realized with the help of Rational Rose UML-editor.

Let's start Rational Rose application, add into the model two classes: "Author" and "Book" (bases of work in Rational Rose are described in Chapter 1). We shall add new attribute into "Author" class. For this purpose, we shall enter a window of its option, having cluck twice on the class image, and select "Attributes" bookmark (see Fig. 4.3).

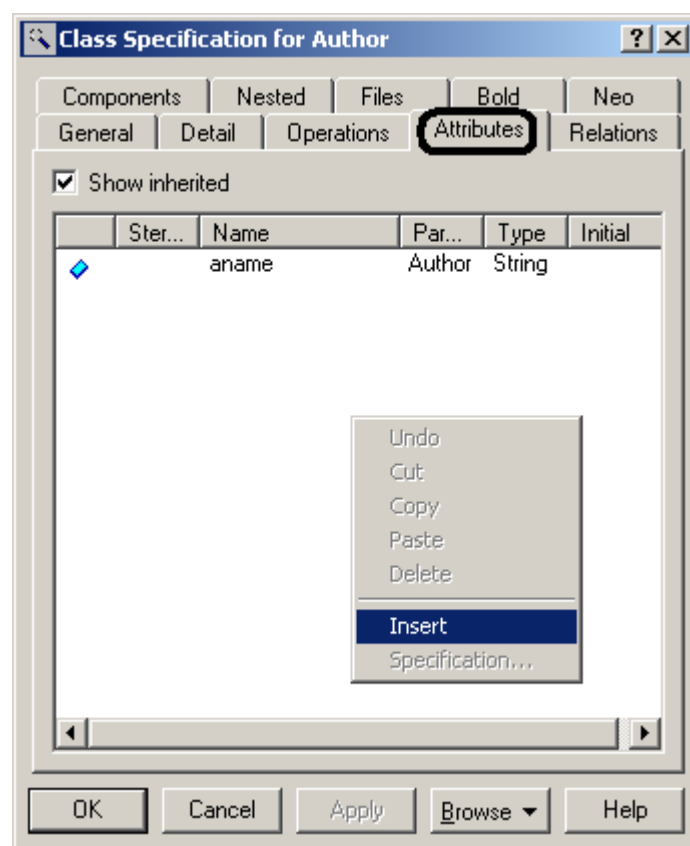


Fig. 4.3. New attribute addition

We shall call dropdown menu by the mouse right button, then select "Insert" item, following which in the attributes list new record with "name" attribute default name will appear. Having cluck twice on this record, we shall enter a window of attribute adjustment, where we shall set "aname" name to it, and then set "string" type using dropdown list (see Fig. 4.4).

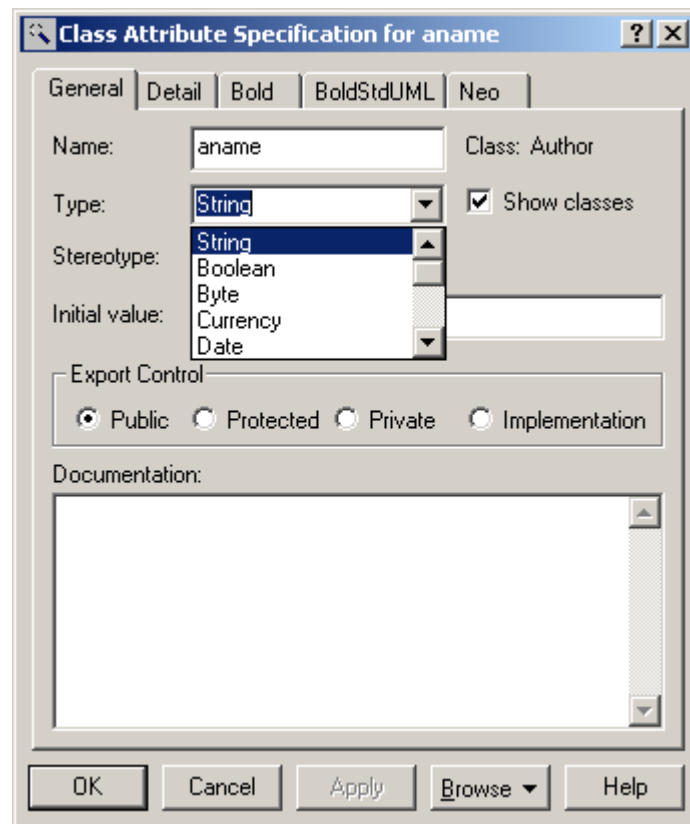


Fig. 4.4. Attribute setting

Similarly we shall add “btitle” string attribute into “Book” class.

Let’s link classes by association, and, having cluck twice on association line, we shall pass to the window of association settings. On “General” bookmark we shall enter roles names of our association: “writes” and “byAuthor”, as Fig. 4.5 shows.

Association Specification for Untitled

Role B General | Role A Detail | Role B Detail | Bold
 BoldStdUML | Neo | Bold A | Neo A | Bold B | Neo B

General | Detail | Role A General

Name: Parent: Logical View

Stereotype:

Role A: Element A: Book

Role B: Element B: Author

Documentation:

OK Cancel Apply Browse Help

Fig. 4.5. Association setting

Further we shall pass on “Role A Detail” bookmark for the first role of association adjustment. Let’s set this role multiplicity as follows “1.. n” (see Fig. 4.6).

Association Specification for Untitled

BoldStdUML Neo Bold A Neo A Bold B Neo B

General Detail Role A General

Role B General Role A Detail Role B Detail Bold

Role: writes Element: Book

Constraints

Multiplicity: 1..n ☒ Navigable

☐ Aggregate ☐ Static ☐ Friend

Containment of Author

☐ By Value ☐ By Reference ☒ Unspecified

Keys/Qualifiers

Name	Type

OK Cancel Apply Browse Help

Fig. 4.6. The first role setting

Similarly we shall set the second role, turning out to “Role B Detail” bookmark (see Fig. 4.7).

Association Specification for Untitled

General Detail Role A General
 BoldStdUML Neo Bold A Neo A Bold B Neo B
 Role B General Role A Detail Role B Detail Bold

Role: **byAuthor** Element: **Author**

Constraints

Multiplicity: **1..n** ☒ Navigable
☐ Aggregate ☐ Static ☐ Friend

Containment of Book
☐ By Value ☐ By Reference ☒ Unspecified

Keys/Qualifiers

Name	Type

OK Cancel Apply Browse Help

Fig. 4.7. The second role setting

In result we shall get the simple model represented in Fig. 4.8. We shall save the created model in a file, for example, “lib.mdl”.



Fig. 4.8. Example of simple model created in Rational Rose

Let's pass to Delphi environment and create the simple project consisting of one form. From <BoldHandles> bookmark we shall place BoldModel1, BoldSystemHandle1, BoldSystemTypeInfoHandle1 components on the form, and adjust them similarly to the simple application setting, which was investigated previously. From <BoldMisc> bookmark we shall place on the form BoldUMLRoseLink component, meant for link with

RationalRose model, and then we shall set in its FileName property a file name of the model saved by us – “lib.mdl” (see Fig. 4.9).

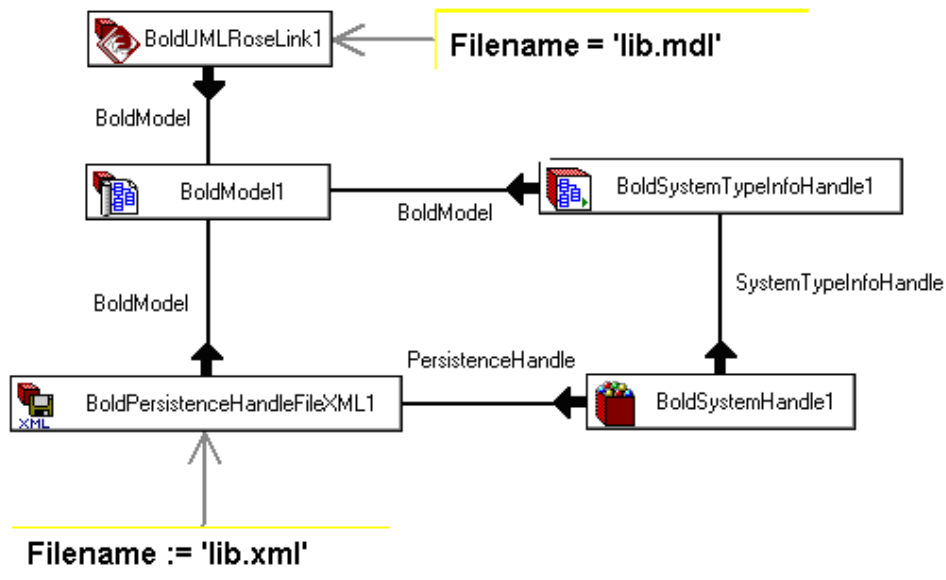


Fig. 4.9. The application Bold-components setting

Now everything is ready for Rational Rose model import into Borland MDA environment. For this purpose, having click twice on BoldModel1 component, we shall open Bold built-in UML-editor, and we shall start import pressing the icon second on the right with an arrow in the toolbar (see Fig. 4.10).

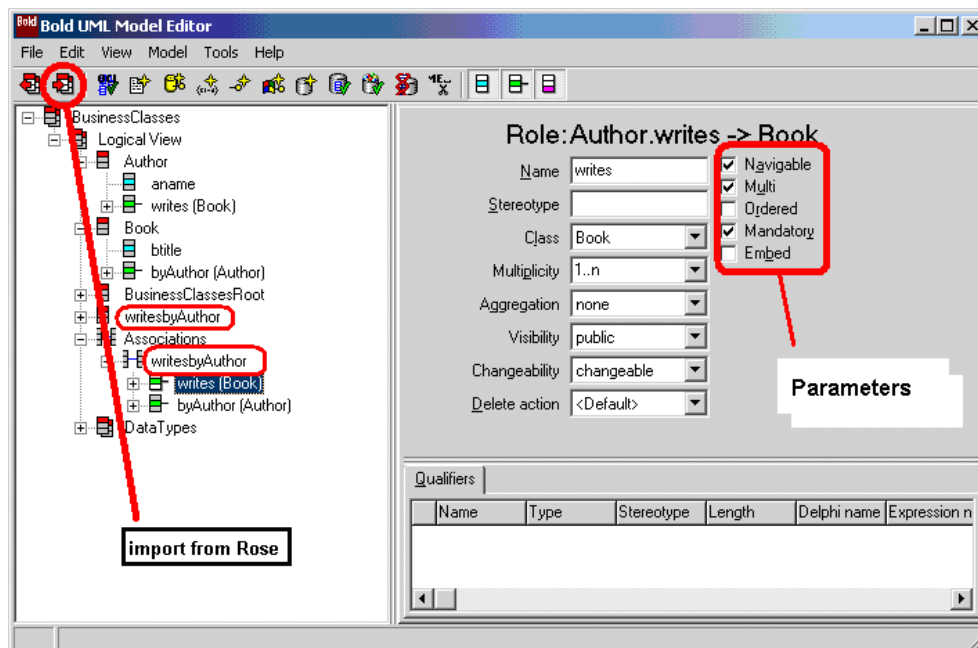


Fig. 4.10. Rational Rose model import into Delphi environment

There will be a window with warning request regarding import confirmation (see Fig. 4.11). Necessity of such warning is caused by the fact that the current model will be replaced by the new one as a result of import operation, therefore the information on the current model can be lost. In case of need this model should be saved beforehand, or this can be made at occurrence of the given warning window, having refused from import.

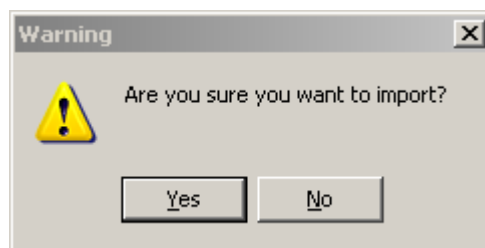


Fig. 4.11. Model import warning

After confirmation import from Rational Rose in Borland MDA environment will be carried out. At that if Rational Rose program has not been started preliminary, there will be its automatic start. Import operation for large models containing of several hundreds of classes can take several minutes. The course of operation can be observed in status line of the built-in editor of models. After import completion it is useful to check up a correctness

of the imported model. It can be made, having chosen in the main menu of models built-in editor Tools ► Consistency Check item. Thus check of sufficiency and consistency of the information set in UML-model will be made. As a consequence of validation the window with results (see Fig. 4.12) will be shown.

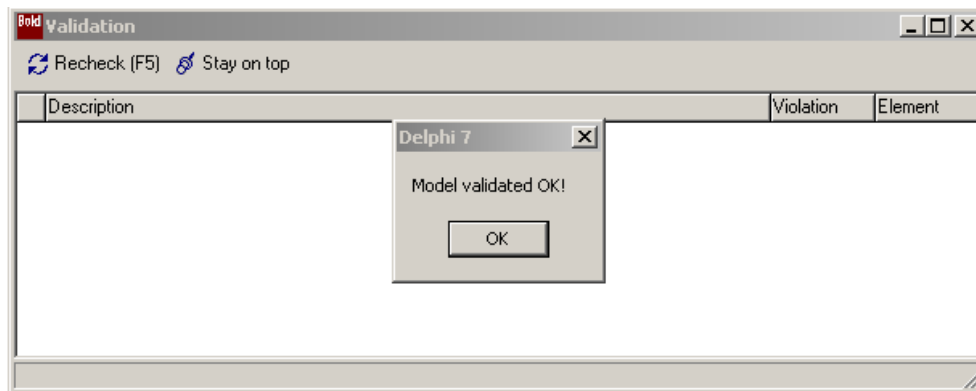


Fig. 4.12. Window with model validation results

Now it is the right time to illustrate advantages which are given with Rational Rose and Borland MDA interoperability. If we look closely at the imported model structure we will find out that the UML-model has changed as a result of import. So, the initial model (see Fig. 4.8) contains two classes, and model imported in Bold includes three classes, the new class named “writesbyAuthor” has appeared (see Fig. 4.13 where two objects are marked by frames: association and class created automatically with “writesbyAuthor” name).

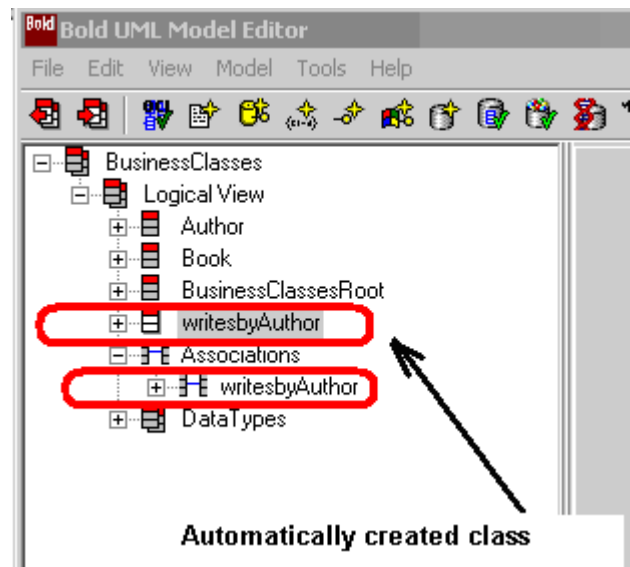


Fig. 4.13. Boldified model

The reason for this class appearance was already discussed in the previous chapter: it is necessary for realization of “many-to-many” relation between “Author” and “Book” classes. But, when we created model in Bold built-in UML-editor, it was necessary to fulfill additional operations for such class creation. Now we can make sure, that Borland MDA and Rational Rose interoperation “is so intellectual”, that it does not restrict itself to information transfer on model elements (classes, attributes, associations).

In our case at import the specified new class has been automatically added. If we set “1” multiplicity value at least for one role in the model, and realize import again, we will make sure easily, that the intermediate class will disappear. Thus, as it was already mentioned, in some cases Borland MDA is capable to add necessary elements in model without assistance. Such operation is named “*models boldification*”, and the inverse one – *unboldification*. In case of need in the Bold-editor it is possible to see both boldified model (see Fig. 4.13) and initial one, for this purpose it is enough to set corresponding parameters in Tools ► Boldify Model or Tools ► Unboldify Model menu. At that unboldified model is similar to the initial one by the structure and also contains only two classes (see Fig. 4.14).

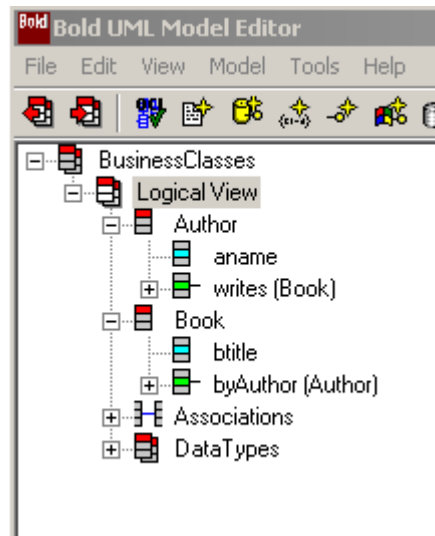


Fig. 4.14. Unboldified model

Thus, the boldified model differs from the initial one by presence of the additional objects created automatically by Borland MDA environment.

Let's keep in mind, that in this case forced boldification of models has been caused by the fact that Borland MDA model objects are saved in relational databases where there are certain constraints – in our case two classes are united by “many-to-many” relation, therefore for saving these classes objects in RDBMS the additional binding table is necessary.

And if we imagine the model containing dozens and hundreds of classes, advantage of using program interoperability of Borland MDA and Rational Rose is that all such binding tables (or rather, classes for associations), presented in “many-to-many” relation, will be automatically generated, and at model creation we don't need to take care of it. That is at import models boldification is carried out automatically. We shall remind, it is possible to boldify the model forcedly from the built-in text UML-editor as we have made in the previous chapter. It is easy to make sure of it if we carry out the following actions:

- ☐ Deleting writesbyAuthor class from model;
- ☐ Checking up model correctness (Tools ► Consistency Check menu), having received the error message at that (see Fig. 4.15);
- ☐ Canceling boldification (Tools ► Unboldify Model);
- ☐ Carrying out boldification (Tools ► Boldify Model).

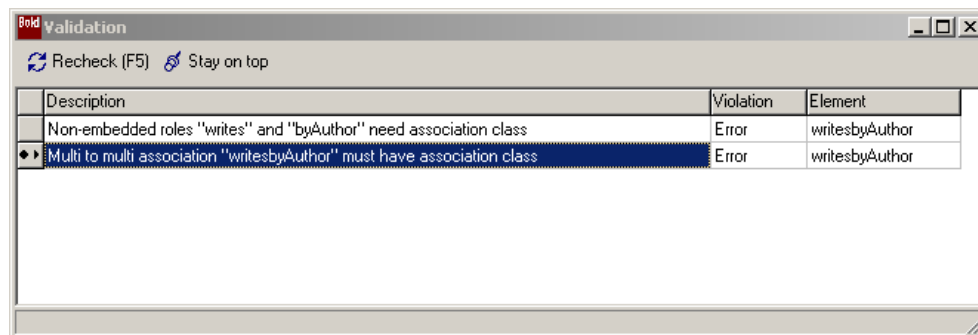


Fig. 4.15. Model verification errors

As a result it is easy to make sure, that writesbyAuthor additional association class deleted by us is created again, and the model becomes correct again.

Model Tagged Values Setup in Rational Rose

Let's return to Rational Rose and modify our model a little, namely, we shall use national-language (for example, French) names of classes, attributes and roles. Let's save the new class diagram in a file, for example, "libfr.mdl". Now we shall begin model adjustment. For this purpose we shall use some tagged values of the model, access to which is provided directly from Rational Rose editor. We shall click twice on "Author" class, turn to "Bold" bookmark, and modify the following tagged values (see Fig. 4.17):

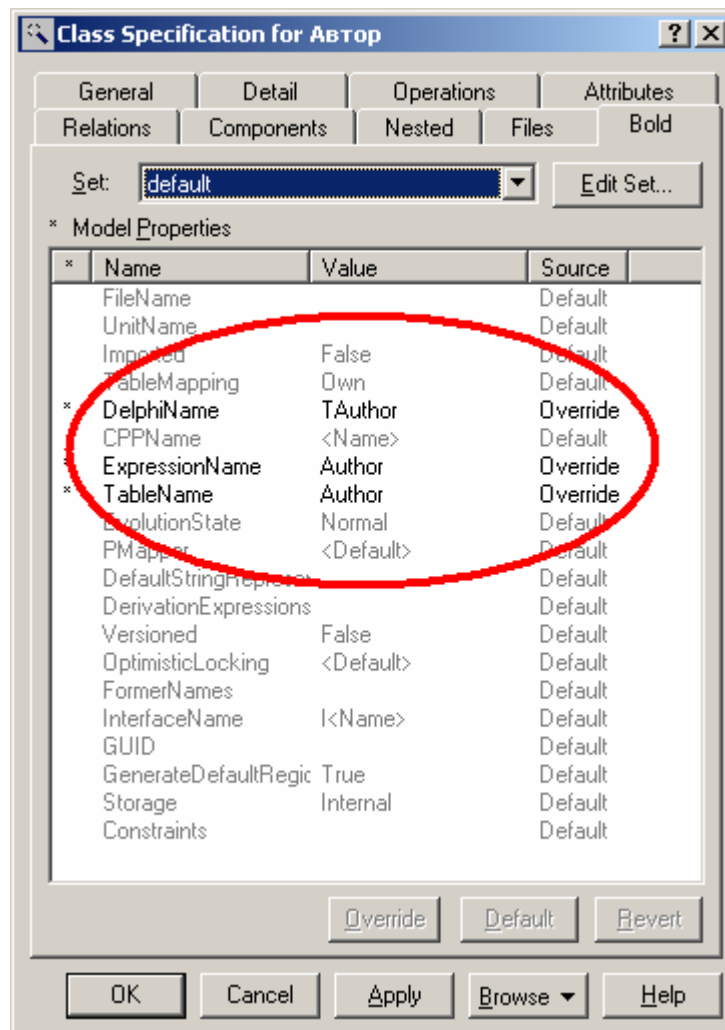


Fig. 4.17. Tagged values adjustment in Rational Rose

- ❑ DelphiName – we shall set TAuthor value; this tagged value is responsible for UML-model class name transformation into the class identifier used in Delphi environment;
- ❑ ExpressionName – we shall set Author value; this tagged value is responsible for model class name transformation into the object name used for class designation in expressions in OCL;
- ❑ TableName – we shall set Author value; this tagged value is responsible for model class name transformation into the name of DBMS table used for information storage on the given class objects.

By default three above-stated tagged values look like T<Name>, <Name>, <Name> accordingly, that means automatic substitution of corresponding name of model class

instead of <Name>. At that in our case French-language identifiers would be generated, that is inadmissible, for example, for “nom-prénom” attribute.

For solving the problem of automatic transformation of model names from national codings in Bold structure there are special means of names mapping which activation is provided, in particular, by NationalCharConversion project tagged value. However generally for their involvement it is necessary to recompile Bold for Delphi source. Other ways of this problem solution are also feasible, for example, by means of Rational Rose – creating a special script in BasicScript language.

Naturally, it is possible to use in the class diagram only English-language names then editing the tagged values described above is not required.

In the same way we shall edit classes attributes – instead of Nom-prénom and Dénomination French-language identifiers we shall set Nom-prenom and Denomination accordingly. (For editing tagged values for attributes it is necessary to turn to specification window of the given attribute and pass to Bold bookmark).

Now we shall save model, return to our Delphi application, change FileName property of BoldUMLRoseLink1 component to “libfr.mdl” new model file name, and then carry out import operation. After that we shall check model correctness (Tools ► Consistency Check menu) and make sure, that Borland MDA environment has sensed all the changes.

At that all classes and attributes names have remained French-language after import though on the panel of additional parameters (it can be called in Bold-editor by pressing Ctrl+A) we shall see the values changed by us. If we repeat actions on creating the simple application, described in the previous chapter, it is easy to see, that in all tables columns headings, and also on autoforms, all designations become French-language as they are taken from names of classes, attributes and application model roles.

Model Tagged Values Setup in the Built-in UML-Editor

For model tagged values setup it is necessary to mark in model tree in Borland MDA built-in editor the required element (class, attribute, association, role) and then to select in <Tools ► Edit tagged values> menu (or simply to press a combination of keys <Ctrl+T>). At that parameters editor window will appear, on which <Bold> bookmark all tagged values concerning to the chosen model element are presented.

For editing concrete parameter it is necessary to mark it in the list of parameters, and enter new value in the tag-editor right window. Besides it is possible to delete tagged values or to add new (user-defined) parameters in the tag-editor. Tagged values are available both during designing the application, and during its run.

As it was said earlier, sets of tagged values are linked to model hierarchical structure layers, and for each layer of this structure there is its own set of such parameters. Below in Table 4.1 the brief description of functional purposes of some parameters is represented. Some parameters (DelphiName, ExpressionName, etc.) have been described above, and they are not given in the table.

Table 4.1. Basic Tagged Values Composition and Function

Tagged Value	Function	Model Elements
DefaultStringRepresentation	Determines string representation for class objects, specifically when mapping on forms, autoforms, grid column titles. Is specified by OCL-expression.	Class
FileName	Defines name of the file containing class operations code, when code generating.	Class
InitialValue	Initial value given automatically to the attribute when calling object constructor.	Class Attribute
Derived	Indicates that the given object value is “derived” according to other objects data. Rules for derivation are set either by OCL-expression or in program code.	Class Attribute
Visibility	Sets “visibility” of object at generation of class property in program code.	Class Attribute
AttributeKind	Kind of attribute. If value=BOLD the attribute is Bold-attribute; if value=Delphi, then it is Delphi-property. In the latter case the information on object is unavailable during runtime. Default=BOLD.	Class Attribute
Length	Length of attribute. Has a value at generation of tables string fields of some DBMS, having constraints on string length. Default=255.	Class Attribute
AllowNULL	Specifies, whether NULL (empty) values are legitimate for attribute. Default=False	Class Attribute
DerivationOCL	OCL-expression for “derived” attributes (see earlier in this table).	Class Attribute
DelayedFetch	If the value is TRUE, it specifies that values of the given attribute should not be called from persistence layer during loading class object. In this case values will be loaded at the first reference to attribute.	Class Attribute
Persistent	If it is TRUE, then attribute value will be saved at persistence layer. For derived-attributes this parameter is ignored.	Class Attribute
Ordered	If it is TRUE, then the role is ordered. At that the additional field for the class table on the opposite association end will be created automatically.	Association Role

DeleteAction	Defines type of action when an attempt is made to delete the object while it has related objects. Possesses the following values: Allow – the object is deleted; Prohibit – program exception is generated; Cascade – The related objects will be deleted as well.	Association Role
--------------	---	------------------

A number of tagged values are specified for adjustment of the uppermost level of hierarchy – a model level. Some of them are represented below in Table 4.2.

Table 4.2. Composition of the Basic Tagged Values of the Model

Tagged Value	Function	Default
ModelName	Specifies the name of the model.	"BusinessClasses"
RootClass	Specifies the name of the ultimate superclass for all classes in model. If this value is empty, BusinessClassesRoot is used as rootclass.	""
GUID	Specifies the global unique identifier of Type Library when generating interfaces. If this value is empty, a random GUID will be assigned.	""
TypeLibVersion	Specifies type library version when generating interfaces.	""
UnitName	Specifies the name of program module containing code for model classes. To be used when code generating.	"BusinessClasses"
ImplementationUses	Sets comma-separated list of unit-names for including into IMPLEMENTATION-section of the generated program module.	""
InterfaceUses	Sets comma-separated list of unit-names for including into INTERFACE-section of the generated program module.	""
UseGlobalId	Specifies need for generating GUID for each object saved in the database.	TRUE
UseXFiles	Specifies need for generating additional table keeping information on each object that has ever existed in the database.	TRUE

UseModelVersion	Sets number of model version. To be used by additional means of versions support.	0
-----------------	---	---

Exporting Model from the Built-in Editor into Rational Rose

After model adjustment in the built-in editor there is a possibility to export it in Rational Rose. For this purpose, as well as at import, it is necessary to set BoldUMLRoseLink component according with model file, and export model by pressing the leftmost button with an arrow in the toolbar (see Fig. 4.23),

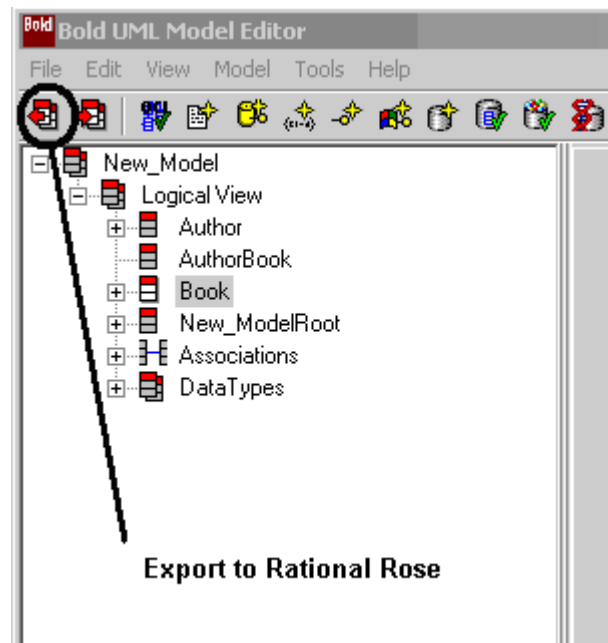


Fig. 4.23. Model export in Rational Rose

or set in menu <File ► Export via Link>. At that if Rational Rose application has not been started beforehand, there will be its automatic start. At model export all tagged values are completely kept. Thus, after tagged values setup in Bold for Delphi environment, it is very convenient to export again the refined model in Rational Rose, and save it in file. It is also possible to create UML-model completely in the built-in Bold-editor, and then export it in Rational Rose editor (for example, into the empty model created beforehand). However at that it is necessary to keep in mind, that after such export the window of class diagram in Rational Rose will look empty. It is obvious, as in model format of the built-in text Bold-editor there is no information on geometrical layout of classes, their sizes, color, etc., while in Rational Rose model format all this information is kept. But we can get out of such a

difficulty, and in a simple way. After such export Rational Rose model tree (see Fig. 4.24) contains all model elements (classes and associations).

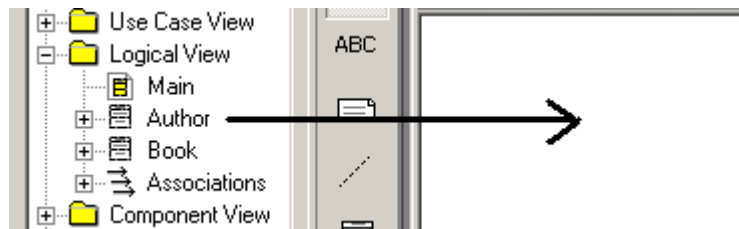


Fig. 4.24. Model elements drag transfer into class diagram

It is enough “to drag” these elements by means of the mouse onto the empty area of the class diagram. Thus graphic presentations of classes will be created automatically, and for the linked classes lines-associations will appear automatically as well. After saving such model in Rational Rose it will be possible to work with it further as usual.

Other Ways of Models Import/Export

For interoperability with Rational Rose there is also another way of UML-models import and export. For these purposes we can take advantage of means of opening and saving models files in the built-in Bold-editor. If we select File ► Open File main menu items and specify Rational Rose model file (with filename extension *.mdl) in the window of file option, in this case Rational Rose application will be also started automatically (if it has not been started earlier), the specified file will be loaded into this application, and then an import into Borland MDA environment will take place automatically. An advantage of such way of import is absence of necessity of using and adjusting BoldUMLRoseLink component.

NOTE

All import and export operations without exception, realizing an exchange of models with Rational Rose CASE-system require obligatory preliminary installation and adjustment of Rational Rose application.

In a similar manner, having created or having edited model in the built-in Bold-editor, it is possible to cause File ► Save File As ... main menu items, specify a filename of a model (necessarily existing!) and thus carry out model export in Rational Rose. At that only one, but essential difference from export by “nominal” way exists (using BoldUMLRoseLink

component). Before such “export through saving” it is necessary to carry out “unboldify” operation (menu Tools ► Unboldify Model), i.e. to remove from the model the additional elements automatically created by Bold environment. Otherwise, if to save boldified model, all “intermediate” classes also will be exported into Rational Rose.

Bold can save and load the information on model not only in files of Rational Rose format (.mdl), but also in own format (.bld), in XMI format, and also in ModelMaker format (.mpb). For using such capabilities it is necessary to set the following steps from the menu of the built-in editor: File ► Open File, или File ► Save File as...

Models Built-in Editor Tools

In this section additional tool means of the models built-in editor will be considered.

Means of Model Properties Adjustment

The panel of model general properties adjustment is mapped when selecting the uppermost element in models tree (see Fig. 4.25). This panel represents a set of the tags-switches specifying the basic properties of the model. We shall briefly consider purpose of some of them:

- ❑ Use X Files – this parameter sets necessity of generation by the environment of the additional table intended for storage of the information on all objects, ever existing in object space. It should be switched on when using separately delivered product-extension “Object Lending Library”, providing data exchange between several databases. Also it is necessary to switch it on when involving UseTimestamp and Use global id parameters (see below);
- ❑ Use timestamp – it is necessary for generation of an additional column in the table which presence is determined by the previous parameter;
- ❑ Use global id – sets necessity of generating 128-bit global unique identifiers (GUID) for each element in a database. Access to such GUID is possible from the program of the application;
- ❑ Use readonly – sets necessity of generating “BOLD_READONLY” additional column for the DB main table of each model class.

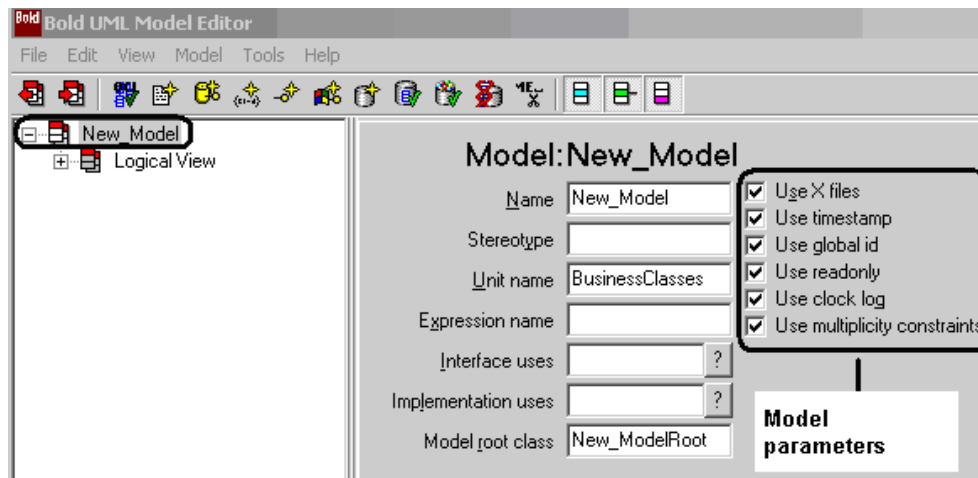


Fig. 4.25. Panel of built-in editor model settings

At real work the tags described above can be left in a condition by default.

Means of Class Attributes Adjustment

For attributes properties adjustment the set of parameters is used being mapped in the built-in editor when selecting class element-attribute in model tree (see Fig. 4.26). Thus the following parameters-properties are available:

- ❑ Persistent – sets, whether the given attribute is saved in database; if the given tag is not marked, the attribute will exist only during the application run;
- ❑ Allow null – shows, whether “empty” value of the given attribute is allowable;
- ❑ Derived – determines the derived attribute (see Chapter 5). Derived attributes are not saved in a database;
- ❑ Delayed fetch – sets the “postponed” call of the given attribute. By default Bold environment loads all object attributes from a database. If there are “large” elements (for example, images, BLOB-objects) in such attributes structure, then for operating speed increase it is expedient to mark the given tag for such attributes. Thus their loading will be carried out at the first reference to the given attribute (for example, at activation of a window in which the image should be displayed).

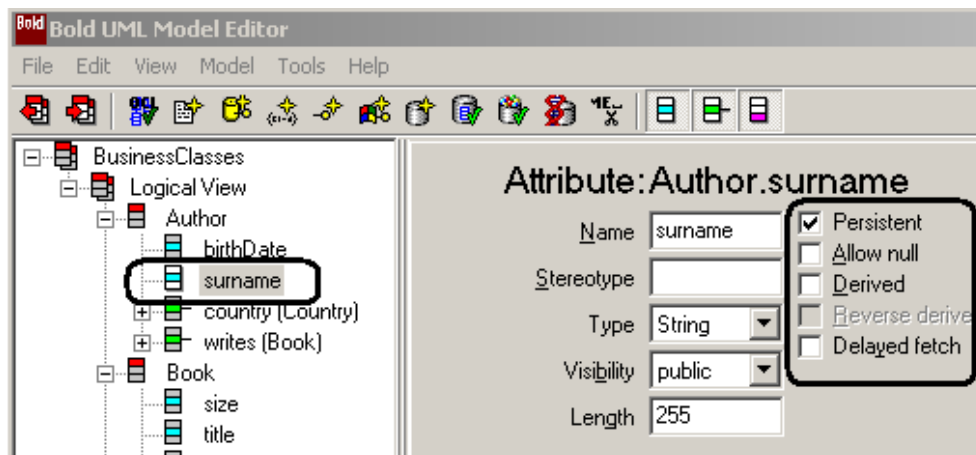


Fig. 4.26. Means of class attributes adjustment

Association Roles Adjustment

When selecting any association role in the built-in editor model tree, on the right there is a set of parameters for its properties adjustment (see Fig. 4.27).

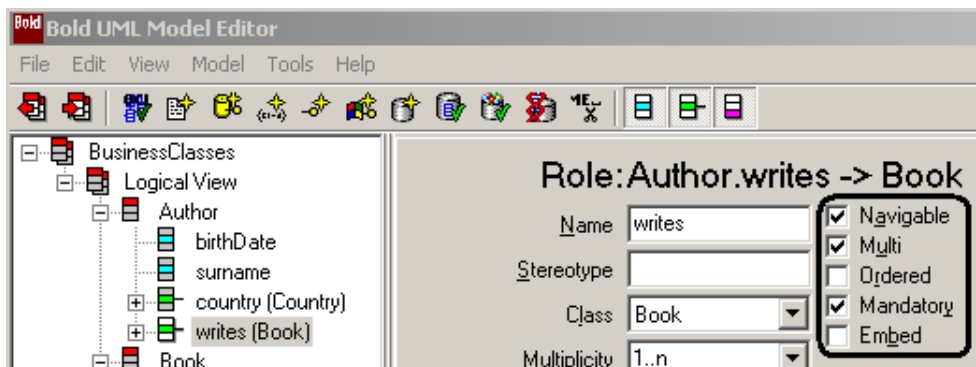


Fig. 4.27. Means of association adjustment

The following concern to them:

- Navigable – means that elements of the given role can be accessible to look-up. For “writes” role represented in Fig. 4.27, the marked tag means, that at navigation over the model we can, having begun from “Author” class context, receive the list of the books written by him. If we remove the mark from the given tag, we shall receive the unidirectional association (see Fig. 4.28). That is in this case we can look through authors of the concrete book, but we cannot receive the list of the concrete author books.

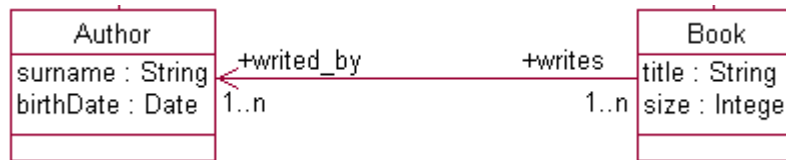


Fig. 4.28. Unidirectional association

We should note, that if the developer has cleaned tags for both roles of association, Bold environment will consider this association as not directed, providing bilateral access to roles elements.

- ❑ Multi – is set automatically in case of role multiplicity more than 1;
- ❑ Ordered – sets the ordered role; thus in the corresponding table of a database the additional column keeping the information on the order of the given role elements will be automatically generated;
- ❑ Mandatory – sets “Mandatory” of a role and is set automatically at the given role multiplicity more than 0;
- ❑ Embed – sets “embedding” the role in the given class (similarly to external keys in tables of relational DB). The parameter can be set only for roles of single multiplicity. If both roles of association are plural, Bold will generate the additional binding table in the database (see also Chapter 3).

Summary

In this chapter we have got acquainted with bases of application model creation in Rational Rose graphic UML-editor and in the models built-in editor. It is quite obvious, that when creating large models, graphic representation visual methods will take advantage compared to the text description. Besides as we made sure, at model import from Rational Rose its boldification is carried out automatically, that also allows not thinking about secondary elements when designing. And, at last, simplicity of access to Borland MDA tagged values from Rational Rose provides necessary flexibility at application model adjustments. It is shown, that presence of tagged values is caused by necessity of tweaking model and its adaptation to Delphi development environment. From this point of view tagged values play a role of platform specific model (PSM) elements.

Chapter 5. OCL – Object Constraint Language

General Information

Object Constraint Language (OCL) appearance has been caused by necessity of description formalization of conditions and constraints, imposed on elements of the UML classes diagram. Such conditions can be formulated also in a natural language, which, however, is not strict and formal, and allows ambiguous interpretation, and as a result it cannot be used in this aspect when creating UML-models. OCL was created as the formal text language complementing graphic capabilities of UML. In this sense OCL is a part of UML. At the same time it is essentially impossible to change model elements (classes, attributes, relations) by means of OCL. OCL also is not the programming language, that is does not allow to create the program from its operators, or to describe any operations execution logic. OCL represents the formal language based on expressions. Any OCL expression returns some value.

OCL has been developed in IBM Corporation, in 1997 the language specification Version 1.1 has been released; such companies as Rational, Microsoft, Oracle, Hewlett-Packard and some other have taken part in its development and coordination. Now OMG Consortium coordinates development of new OCL specifications.

OCL Role in Borland MDA

OCL plays extremely important role in Borland MDA, carrying out the following basic functions:

- ❑ Navigation on model elements (classes, attributes, associations);
- ❑ Conditions and constraints assignment for the model elements;
- ❑ Expressions assignment for derived attributes.

Navigation over the model, provided by means of OCL in Borland MDA, allows carrying out the flexible and powerful mechanism of “inquiries” to the application *object space*. The concept of object space will be described later, now it is enough to tell, that it covers realization of all elements of model during application run. Such “OCL-inquiries” basically are capable to replace completely SQL, which is usual for developers of databases; at that these inquiries are visual, laconic and powerful. Besides taking into account OCL platform independence, such inquiries are universal, and they are not bound absolutely to the concrete DBMS used in the application.

Model for Studying

For studying OCL capabilities we shall examine the following UML-model (see Fig. 5.1). We shall consider, that this model describes the conditional application for work with a library catalogue. The model contains the following classes:

- ❑ Country – has got “name” text attribute (country name);
- ❑ Author – has got “surname” text attribute and attribute of “date” type – “birthDate” (date of birth);

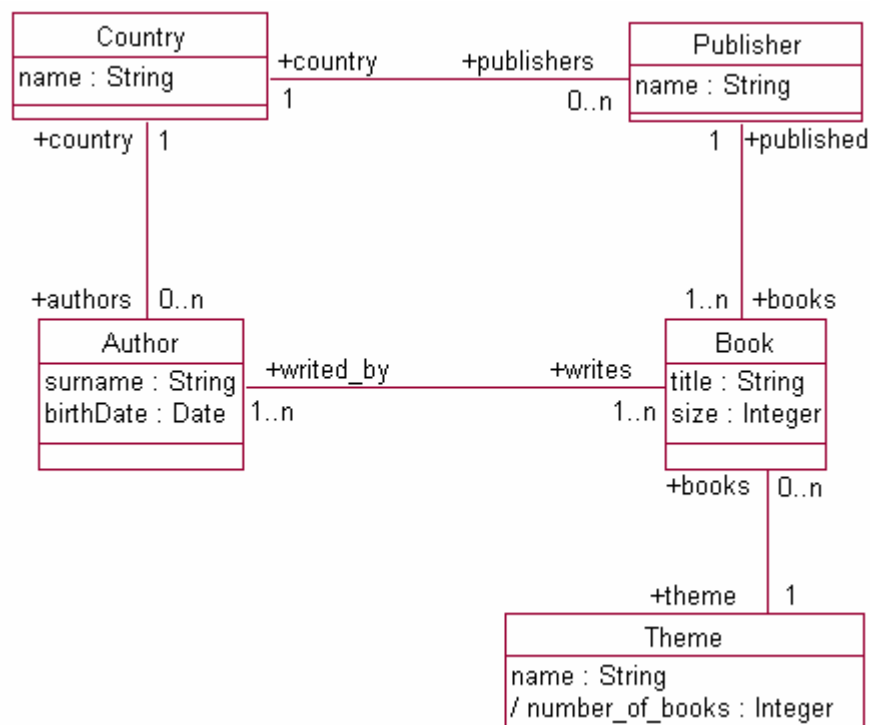


Fig. 5.1. UML-model under study

- ❑ Publisher – has got “name” text attribute;
- ❑ Book – has got “title” text attribute and “size” integer attribute (number of pages);
- ❑ Theme – has got “name” text attribute and “number_of_books” derived integer attribute.

Relations-associations contained in model allow formulating the following basic business-rules determined by our model:

- ❑ Each author can write one or several books; each book can be written by one or several authors;
- ❑ Each author is the citizen of one country;
- ❑ Several authors or no one author can live in each country;
- ❑ In each country there can be several publishing houses or no one;
- ❑ Each publishing house belongs only to one country;
- ❑ Each publishing house can issue one or several books;
- ❑ Each book can be issued only by one publishing house;
- ❑ Each book can relate only to one subject;
- ❑ There can be several books belonging to one subject, or there can be no one book on the given subject.

Access to Classes and Attributes from OCL

For gaining all instances (objects) of model concrete class in OCL `allInstances` operation is used, thus, for getting the list of the countries in our model it is possible to use the following OCL-expression:

`<Country.allInstances>.`

We can see in this example, that OCL uses the so-called “dot-notation”, that is dot symbol between expression elements. The given expression for Country class returns a set of objects of “Country” type. If we want to receive a set of countries names, we should take advantage of the following expression: `Country.allInstances.name`, i.e. specify a name of class concrete attribute in OCL-expression.

Base Elements and Concepts

Data Types

OCL, being formal language, supports various data types. Their list and necessary explanations are represented in Table 5.1.

Table 5.1. Data types in OCL

Type	Type Description
Integer	Any integer number
Real	Any real number
Logical	“True” or “False”
String	Any number of symbols

Comparison Operations in OCL

OCL supports the following comparison operations used in generation of various conditions:

- `=` (equal);

- < (less);
- > (more);
- <= (less or equal);
- >= (more or equal);
- or;
- and;
- not;
- xor (“exclusive OR”);
- implies (“one-way AND”).

We shall consider more in detail last operation – “implies”, as it is not widely known. Implies acts as peculiar operation of “and” type, but operates “in one way” (see Tab. 5.2). Its sense in OCL is approximately the following:

“If the result of the left part of expression is true, then the result of the right part also should be true. If the left part of expression is false, then the result is always true”.

Table 5.2. «Implies» operation values variants

	A=true B=true	A=true B=false	A=false B=true	A=false B=false
A implies B	True	False	True	True
B implies A	True	True	False	True

Return Expressions Types

As a result of using some OCL-expression results of the following types can be obtained:

- ❑ Metadata – information on types, OCL also provides some capabilities concerning types cast;
- ❑ Objects – model classes instances;
- ❑ Types of model attributes (integer, string, etc.)

Besides the result can be also represented as a set from elements of each type listed above - *collection*. At that each collection can contain inside itself only one-type elements.

Arithmetic Operators

OCL supports the standard arithmetic operators which structure and examples of use are represented in Table 5.3.

Table 5.3. Arithmetic operators in OCL

Operator	Examples <i>[result]</i>	Description
----------	--------------------------	-------------

+	2+9 [11] '2'+ '3' ['23']	Addition
–	19 – 2 [17]	Subtraction
*	Book.allInstances – >size*2 [Doubled number of books]	Multiplication
/	6/2 [3]	Division
()	(3 – 2)*4 [4]	Parentheses
.round	(Book.allInstances – >size/ Author.allInstances – >size).round [Rounded average number of books, written by one author]	Rounding the numeric value to the nearest integer
.floor	375.8.floor [375]	The integer portion of the numeric
.abs	– 23.abs [23]	The absolute value

Collections

“Collection” concept can be found in OCL frequently enough. The collection is the type of the data determining some set of elements of one type. In OCL three basic kinds of collections exist:

- ❑ Set – determines unordered set of elements without repeating elements;
- ❑ Bag – multiset – determines unordered set of elements allowing repeating elements
- ❑ Sequence – determines ordered set of elements without repeating elements.

For applying operations to collections in OCL the special operator of collection operation designated as “– >” is used.

For example, <Book.allInstances – >asSet> OCL expression will return set-collection of objects of “Book” type, not containing identical books. And <Author.allInstances – >size> expression will return quantity of all authors, probably, with repeating elements in the list. Here “size” operation carries out function of getting number of elements of the collection to which it is applied.

Navigating Model

Using OCL it is simply enough to provide access to model elements with the help of relations-association between classes. At that one class can act as OCL-expression *context*, and elements of other classes present a result. We shall note, that “context” concept in this case means the model element acting as a source of the information for OCL-expression. The reserved word <self> is used in OCL for the current context designation.

Let's consider, for example, the following expression: Country.allInstances.authors. In this case “Country” class acts as a context-source, however OCL-expression considered further

includes “authors” role name of the association linking “Country” class with “Author” class. Therefore result of expression in this case will be the collection of objects-instances of “Author” type. It is possible to construct similarly such OCL-expressions as `Country.allInstances.publishers.books` or `Book.allInstances.wrote_by.country`.

Thus, it is clear, how the navigation is carried out for UML-models – beginning from a concrete class and consistently finishing writing in OCL-expression roles names of the opposite ends of associations-relations linking the next in chain class with the following, basically it is possible to get the results concerning to any model class. A necessary condition is presence of associations between classes in the chain constructed in such a way. And again we can see here active use of “dot-notation”. At that there can be a natural question: what for “->” operator of a collection considered in the previous subsection is used in this case, why is it impossible to use simply “dot-notation”? For answering the question we shall consider the following OCL-expression: `Book.allInstances.size`. In this expression an ambiguity is present caused by names coincidence of “size” attribute and elements number collection operation. Therefore in this case it is not quite clear, what do we mean - number of all books or a set-collection which elements contain number of pages of each book. Collection operation eliminates such ambiguity, and `Book.allInstances->size` expression has a single treatment – number of all books.

From the examples considered above it becomes quite obvious, that capabilities of navigation on model by means of OCL-expressions are defined basically by correct construction of the model structure. For the separate isolated model class which does not have associations-relations with other classes, such navigation is impossible.

In all cases examined above collections that can be gotten by more simple ways are results of OCL-expressions. Really, if we shall consider formally quite allowable `Theme.allInstances.books.published` OCL-expression, from its content it is obvious that the collection containing all publishing houses is a result of such expression. Clearly, that the same result can be obtained simply as `Publisher.allInstances`. So in such view OCL-navigation capabilities look not absolutely clear from the practical point of view. In order to achieve practical sense for such navigation, it is expedient to use special operations for collections, which we shall consider just now.

Collection Operations

Navigating Collection

The given operations are used for moving on a set of the elements forming a collection.

First Operation

It is used for selecting the first element of the collection. For example,

`Book.allInstances ->first.name`

OCL-expression returns the title of the first book as a result.

Last Operation

This operation is used for selecting the last element of the collection.

For example,

`Book.allInstances ->last.authors`

OCL-expression will return the list of authors of the last book as a result, and, for example, such expression as

`Book.allInstances ->last.authors ->first.name`

will return the name the first author of the last book.

At Operation (K – integer)

This operation is used for selecting a concrete element of the collection, which is located on the Kth place in a set. For example, the following OCL-expression

`Author.allInstances - at(3).name`

returns the name of the third author from the list of all authors.

Selecting (Filtering) and Rejecting

Select(<condition>) Operation

“Select” operation is used in OCL for realizing collections filtering, i.e. for selecting subset of elements according some condition.

For example, for selecting all objects of “Country” type except for the country with “Mozambique” name, it is possible to generate the following OCL-expression:

`Country.allInstances ->select(name<>'Mozambique')`

In result we shall get the collection of the countries, which does not contain the mentioned country. Involving OCL capabilities for navigation on model, it is possible to write the following expression:

`Country.allInstances ->select(name='Mozambique').authors.writes`

– and to receive the list of all books, which authors live in Mozambique country. In this example inside parentheses of “select” operation there are attributes belonging to a class, objects of which are filtered (in the given expressions attribute “name” – the name of the country – was used). “Select” operations can be united in chains in order to realize more complex OCL-queries. For example, we shall consider the following OCL-expression:

`Book.allInstances`

- `>select(published.country.name<>'Japan')`
- `>select(theme.name='textbook') ->size`

The number of books issued by all publishing houses, with the exception of the Japanese publishing houses is a result of this OCL-expression, at that only the books concerning to educational subjects are selected. The interesting thing in this expression is the fact of using a condition of selecting “navigational” expressions: “published.country.name” and “theme.name”, containing both roles, and attributes of “final” classes – “Country” and “Theme”. By this example it is possible to estimate flexibility of the approach realized by OCL. If we carry out comparison with SQL it will not be in favour of the last mentioned, and for getting even more simple result we should generate rather bulky SQL-operator, something like the following (in notation conventions):

```
select b.name,b.idpub,b.idtheme,p.idcountry
from book b, theme t, publisher p, country c
```

where b.idpub=pub.id and b.idtheme=theme.id
 and p.idcountry=country.id and c.name<>'Japan'
 and t.name='textbook'

Let's note, that in above-stated OCL-expression "many-to-many" relations between UML-model classes were not used, otherwise it would be impossible to generate similar SQL-query in view of relational model constraints (in this case we should add the binding table artificially). Besides the shown record laconicism, OCL-expression, also gains in obviousness without doubts.

Reject(<condition>) Operation

Reject operation is inverse operation in relation to Select one, i.e. all records satisfying some condition, are excluded from a resulting set of elements. Syntax of Reject and Select operations and rules of their use in OCL-expressions are identical. For example, the following OCL-expressions are completely equivalent:

```
<Country.allInstances - >select(name<>'Mozambique')>
<Country.allInstances - >reject(name='Mozambique')>
```

Operations with Several Sets

Difference(<collection>) Operation

The sense of Difference operation lies in getting the resulting set of values as difference between the original collection and some specified external collection. The given operation uses two collections and requires some external collection of elements to be generated before realization.

For example, we shall consider the following OCL-expressions:

```
Book.allInstances - >select(theme.name='fiction')
Book.allInstances - >select(published.country.name='Russia')
- >difference(collection)
```

The first expression returns the collection of fiction literary works, and the second takes this collection as a parameter, compares it by elements to all books issued in Russia, and returns a difference of these two collections of books, i.e. in result the list of books of the Russian publishing houses, excepting fiction literary works will turn out.

SymmetricDifference(<collection>) Operation

This operation also works with two collections of elements. Its result is presented by the set-collection containing elements that appear in exactly one of the collections. Applying OCL-expressions similar to the previous example

```
Book.allInstances - >select(theme.name='fiction')
Book.allInstances - >select(published.country.name='Russia') -
>symmetricdifference(collection)
```

we shall get in result the collection of books including either fiction literary works issued anywhere except for Russia, or inartistic Russian books.

Intersection(<collection>) Operation

The given operation allows getting intersection of two collections, i.e. a collection of the elements that appear in both specified collections concurrently.

For the similar expressions

```
Book.allInstances - >select(theme.name='fiction')
Book.allInstances - >select(published.country.name='Russia') -
>intersection(collection)
```

we shall get in result the list of fiction literary works issued by the Russian publishing houses.

Union(<collection>) Operation

As it is easy to guess from the title, the given operation returns union of two specified sets-collections. For example, for the following expressions

```
Book.allInstances - >select(theme.name='fiction')
Book.allInstances - >select(published.country.name='Russia')
- >union(collection) - >asSet
```

in result we shall receive the list uniting all fiction literary works plus all books of the Russian publishing houses. For excluding repeating values in the second expression, “asSet” collection operation already familiar to us is applied.

Sorting Collection Elements

OrderBy(<OCL-expression>) Operation

The given operation provides collection sorting by condition specifying with expression-parameter. Any allowable OCL-expression can act as a parameter, i.e. the result of this expression should satisfy the context.

For example,

```
Book.allInstances - >orderby(name)
```

OCL – expression will return as result a collection of the books ordered according their names.

And

```
Author.allInstances - >orderby(country.name)
```

expression will return the list of authors ordered by names of the countries. In the mentioned examples expression-parameters satisfy a context as they are constructed either from attributes of a class above which operation is made, or from a role of the association linked to it and attribute of the bound class. And the following OCL-expression:

```
Author.allInstances - >orderby(publishers.title)
```

is incorrect, as Author and Publisher classes are not linked by the relation of association.

OrderDesending(<OCL – expression>) Operation

This operation is similar to the previous one, there is only one difference that collection - result sorting has reverse order. Collections of authors identical on structure, but ordered by mutually-opposite way, are the results of the following two OCL-expressions:

```
Author.allInstances - >orderby(country.name)
```

```
Author.allInstances - >orderdescending(country.name)
```

Logical Operations on Collections

Includes(<element>) Operation

The given operation returns “True” logic value if the element specified as parameter presents in collection, and otherwise this operation returns “False” value.

For example,

```
Country.allInstances.name - >includes('Russia')
```

OCL-expression result will be true if Russia country presents at the list of all countries, and otherwise the result is false.

forAll(<condition>) Operation

The given operation checks the specified condition-parameter for each element of the collection, and returns “True” logic value only in the case when this condition is true for all elements. If it is wrong at least for one element of the collection, result of the given operation will be “False” logic value. For example, the following OCL - expression

```
Author.allInstances - >forAll( country.name='USA')
```

will return “True” as a result only in the case when all the authors live in USA.

IncludesAll(<collection>) Operation

This operation uses a collection-parameter, and checks, whether each element of the collection-parameter is included into the collection above which operation is carried out. Result of operation is “True” only in the case when all collection-parameter elements without exception are terms of the processing collection. For example, we shall consider the following OCL-expressions:

```
Author.allInstances - >select(country.name='France').books
```

```
Book.allInstances - >select(published.country.name='France')  
- >includesAll(collection)
```

The first expression returns the list-collection of the books written by authors, living in France. The second one selects the list of the books issued by the French publishing houses, and checks, whether all books from the first collection are included into this list. If at least one book of the author-Frenchman has been issued not in France result of the given operation will be “False”.

IsEmpty Operation

Checks, whether the collection is empty (containing no one element), and returns in this case “True” value. If the collection contains at least one element, result will be “False”.

NotEmpty Operation

This is inverse operation with respect to isEmpty operation, and it returns “True” if the collection is not empty. For example,

```
Author.allInstances ->select(name='Smith').books  
->select(theme.name='Textbook') ->notEmpty
```

OCL-expression will be true if the author with “Smith” surname has written at least one book – the textbook.

Collect Operation

Collect operation is rather powerful tool of OCL. As a result of this operation use the new collection is created, which type depends on parameter-condition of the given operation. At that the collection-result represents a set of elements, and each of this sets comes out from an original grouping of an initial collection on the specified parameter. We shall consider, for example, the following OCL-expression

```
Publisher.allInstances ->collect(books ->size).
```

In result we shall receive the list (a collection of integers), which each element represents a number of books issued by concrete publishing house, and number of elements in this list will be equal to the number of all publishing houses. It is easy to see, that without using Collect operation the result will be absolutely different:

```
Publisher.allInstances.books ->size
```

In the latter case result of expression will be the total of books though result of Publisher.allInstances.books expression will be also the collection, i.e. this expression is similar to the following expression

```
Publisher.allInstances ->collect(books)
```

Thus, OCL “interprets” using “dot – notations” in relation to property as Collect operation realization.

Let's consider more complex example. We shall assume, that it is necessary for us to receive the general statistics on all countries, including number of all books issued in everyone country. We shall notice, that in examined model (see Fig. 3.1) the direct association connecting “Country” and “Book” classes is absent. The algorithm of the decision in a natural language can be formulated for the given case approximately so: for the concrete country it is necessary to select all its publishing houses, to get total number of books issued by each publishing house of this country, sum all obtained values for each country, and to repeat this algorithm for all countries. Using Collect operation and having corresponding experience, we shall “translate” easily the natural description of algorithm into OCL, having generated the following OCL-expression solving the problem:

```
Country.allInstances ->collect(publishers ->collect(books ->size) ->sum)
```

We shall carry out detailed analysis of this OCL-expression. Let's consider it "from the end". Books->size expression in context of publishers gives a number of books issued by concrete publishing house. Publishers->collect (books->size) expression returns a set of elements, each of which represents total number of books issued by concrete publisher, and number of elements of this set is equal to quantity of publishers. The following OCL-expression

```
publishers ->collect(books ->size) ->sum
```

provides summation of all elements of the previous expression, i.e. gives total of the books issued by all publishing houses, participating in this expression (here for the first time "sum" OCL-operation of collection elements summation is used, and its sense is obvious enough).

And in order to achieve the situation when these publishers-"participants" belong to the same country every time, one more Collect operation is applied to the countries as it has been done already for publishing houses, and this operation provides "grouping" on publishing houses.

It is necessary to realize precisely, that when forming obtained OCL-expressions we based on specified UML-model. So, for example, if the "Country" class did not have association with "publishers" role we could not write parameter expression for the first Collect operation. In other words, when forming OCL-expressions frequently it is useful and even necessary to apply navigation on UML-model. Due to this example we can see how OCL and UML languages are closely integrated. And at that they supplement effectively each other: UML plays a role of "architect" of the system as a whole, and OCL uses knowledge of this architecture (UML-model) for solving specific problems.

Considering the obtained OCL-expression:

```
Country.allInstances ->collect(publishers ->collect(books ->size) ->sum)
```

It is necessary to note one more time capacity and elegance inherent in language OCL. Also it is possible to ascertain, that OCL expressions, as against many other formal languages, look naturally enough, and at that they are easy-to-read.

Calculations on Collections

Count(<element>) Operation

Returns number of the collection elements satisfying the specified parameter. At that the type of parameter should concur with the base class type of a processing collection. For example,

```
Author.allInstances ->count(Author.allInstances ->select(name='Smith'))
```

OCL-expression returns as a result a number of authors with "Smith" surname.

We shall note, that this result can be obtained by more simple way:

```
Author.allInstances ->select(name='Smith') ->size
```

Sum Operation

This is operation of the collection elements summation, returning numerical value. Use of the given operation was shown in the previous subsection.

MinValue Operation

It is used for getting the minimal value from a collection.

For example,

```
Author.allInstances ->collect(books ->size) ->minValue
```

OCL-expression returns as a result minimum quantity of the books written by one author (at that, it is not known which author has written the least quantity of books, there can be several authors though).

MaxValue Operation

This operation is similar to the previous one except that the maximal value from a collection is returned. On the basis of the previous example of expression for MinValue operation, we shall construct the following OCL-expression:

```
Author.allInstances ->select(books ->size=Author.allInstances  
->collect(books ->size) ->maxValue).name
```

In result there will be the list of the authors who wrote a maximum quantity of books. We shall note, that in this case we do not know an actual type of the resulting expression, whether it will be a line – a name of the author, or the list of lines – names of such authors. Really, if we assume the maximum quantity of the books written by one author equal to 5, the situation can occur, when several authors have written 5 books, and in this case all of them will be included in OCL - expression result. Similar situations, when the type of result is unknown beforehand (value or array-collection) occur quite frequently when working with OCL. Thus, OCL always “consider” such results as collections, irrespective of quantity of elements in result. For bringing result, for example, to string type, it is possible to take advantage of the operations described above: “first”, “last”, and “at”, returning a single element of the collection. So, the following OCL-expression will have string type and will return a name of the first author who has written a maximum quantity of books:

```
Author.allInstances ->select(books ->size=Author.allInstances  
->collect(books ->size) ->maxValue).name ->first
```

Working with Types in OCL

Processing Types for Elements

OclType and TypeName Operations

These operations return type of an operand. Their difference is that “TypeName” operation returns not type itself, but its name as a string.

For example,

`Author.typeName`

OCL-expression will return ‘Author’ string as a result.

***oclIsTypeOf*(<metatype>) Operation**

The given operation returns “True” logic value if the type of an operand concurs with type of parameter. Otherwise “False” logic value is returned.

***ocAsType*(<metatype>) Operation**

This operation carries out resulting an operand to the type specified by parameter. If such transformation of types is impossible, the exception is developed. As a rule it is expedient to use this operation only for classes-successors of some super class.

***safeCast*(<metatype>) Operation**

The given operation has the same sense, as the previous one, but it does not develop exceptions when types transformation is impossible. Instead of this in such situation <nil> will be result of operation.

***superTypes* Operation**

As result the collection of the nearest super classes (direct classes-parents) of an operand is returned. For the development environments, which do not support multiple inheritance (for example, for Borland Delphi), a collection with a single element will be a result of operation.

***allSuperTypes* Operation**

The given operation is similar to the previous one by its functional purpose with the exception of the fact that as result all super classes are returned, but not only direct classes-parents. Such situation occurs at presence in model of heritable classes chains.

***allSubClasses* Operation**

This operation returns a collection of all child classes.

attributes Operation

This operation returns set of attributes of the given class. For example,

`Book.attributes`

OCL-expression will return as a result attributes list of “Book” class: “title, size”.

associationEnds Operation

Returns the list of the associations ends (roles) of the given class. For example, for “Book” class this operation will return the list of roles – “published, written_by, theme”.

Processing Types for Collections

filterOnType(<metatype>) Operation

This operation returns subset, which elements type meets type of input parameter.

Logical Choice Operations

In OCL there is an operation of the logic choice, which allows forming OCL-expression result depending on logic conditions specified by the developer. Concerning syntax this operation is close to similar operators of known programming languages, and has the following view:

```
if <condition>
then <expression 1>
else <expression 2>
endif
```

Result of the given operation is OCL-expression, selected from two probable expressions, proceeding from truth or falsity of condition specified in operation.

Essential difference of the given operation syntax from analogues is mandatory presence of expression 2, i.e. “else...” branch. We shall consider the following two OCL-expressions:

```
Book.allInstances
if (publisher.country.name='Russia') then 'Russian'
else 'Foreign'
endif
```

Let's assume, that the second expression works with a context of the first one (i.e. with a context of “Book” class). In this case as result of work of the second expression we shall get the list of strings containing words “Russian” or “Foreign” depending on whether the book is issued in Russia or in another country. The size of the list obtained will concur with total of books.

It is necessary to have in view that seeming sameness of the considered operation with similar operators of programming languages does not mean that OCL can contain

fragments of program realization. Whatever difficult the content of the considered operation may be in result we always have got simply the concrete OCL-expression instead of changing the program run.

Other Operations

Operations with Strings

OCL has the advanced means of work with strings providing flexible capabilities on processing string items.

Concat(<string>) Operation

Concatenates together an operand and string-parameter. For example, the following OCL-expression

```
Author.allInstances.surname.concat(' author')
```

will return a collection of strings, each of which is formed by a surname of the author and the 'author' string added at the end of each string of the list. We shall note, that this operation is replaced completely with “+” operator of addition, so that the following expression is completely equivalent to the expression considered above:

```
Author.allInstances.surname+' author'
```

pad(<integer>,<character>) Operation

The given operation changes string length (quantity of characters) by addition of a character-parameter in the beginning of the string so that the resulted total string length is not less than specified parameter-integer. If the initial string length is equal or exceeds the integer-parameter, no changes are made.

For example, the following OCL-expression

```
'name'.pad(5,' - ')
```

will return the string: ‘ – – name’, which length is equal to 5.

postPad(<integer>,<character>) Operation

This operation is similar to the previous one; only characters-parameters are added to the end of a string.

For example, the following OCL-expression

```
'age'.postPad(8,'_')
```

will return the string: ‘age_’, which length is equal to 8.

sqlLike(<string>) and sqlLikeCaseInsensitive(<string>) Operations

As a rule the given operations are shared with Select operation, and provide searching the records containing the specified string-parameter. Basically this allows the use of the % (percent) character to act as a wild card. For example, the following OCL-expression

`Author.allInstances ->select(surname.sqlLike('&ac&'))`

will remain in resulting collection only the authors, in whose surnames 'ac' substring is found. In this sense the given operations correspond ANSI SQL Like operator. SqlLikeCaseInsensitive operation is remarkable for carrying out selection without taking into account the characters case.

subString (<integer>,<integer>) Operation

Returns substring consisted from characters of the original string, at that initial and terminating positions of the original string characters are set by input parameters. For example,

`'locomotive'.subString(3,5)`

OCL-expression will return 'com' string.

If the specified number of the terminating character goes beyond the end of the original string, it is ignored and the subset ends at the original string last character, i.e.

`'locomotive'.subString(5,19)`

expression will return 'motive' string.

toLower and toUpper Operation

These operations result all characters of strings-operands to the lower (toLower) or to the upper (toUpper) case.

Operations of Strings Transformation to the Other Types

These operations (see Table 3.4) are intended for transformation of types. When types transformation is impossible exception is developed.

Table 3.4. Types transformation operations

Operation	Type of return result type
strToInt	Integer
strToTime	Time
strToDate	Date
strToDateTime	Date-Time template

Operations with Dates and Times

Date and Time Formats

ISO standard for dates and time representation is used in OCL. For date <#year – month – day> notation is used, for example, #2003 – 11 – 26. For time <#hours:minutes:seconds>

notation is used, for example, #02:23:45. For dates and time the operations of comparison described above are allowable, for example, the following OCL-expression:

```
Author.allInstances ->select(birthDate<#1980 - 01 - 01)
```

will return as a result the collection of authors, who was born after January, 1, 1980.

inDateRange(<date 1>,<date 2>) Operation

The given operation returns “True” logic value if input date is in the range of dates between “Date 1” and “Date 2”, otherwise result of operation will be “False” logic value.

SumTime Operation

This operation applied to the collection, all elements of which have “Time” type, returns total amount of time.

Using OCL

Expressions Assignment for Derived Attributes

OCL is frequently used for forming expressions for derived attributes. For example, in Theme class of considered UML-model there is “number_of_books” derived attribute (number of all books on the given subject). In this case when forming model for this attribute the following OCL-expression can be written down:

```
books ->size
```

At the first reference to the given attribute (for example, when mapping its value) its value will be operatively designed.

Forming Derived Attributes in Rational Rose

For forming derived attributes in the UML-editor of Rational Rose CASE-system it is necessary to create new attribute, and then turn to “Detail” bookmark in the window of the attribute specification (see Fig. 5.2), where it is necessary to make active (to mark) “Derived” tag. Thus we informed the modeling environment that the given attribute is derived. Besides we should generate OCL-expression for the given attribute according to which its value will be calculated. For this purpose it is necessary to proceed to “Bold” bookmark, and select string with “DerivationOCL” parameter in the opened list of tagged values (see Fig. 5.3).

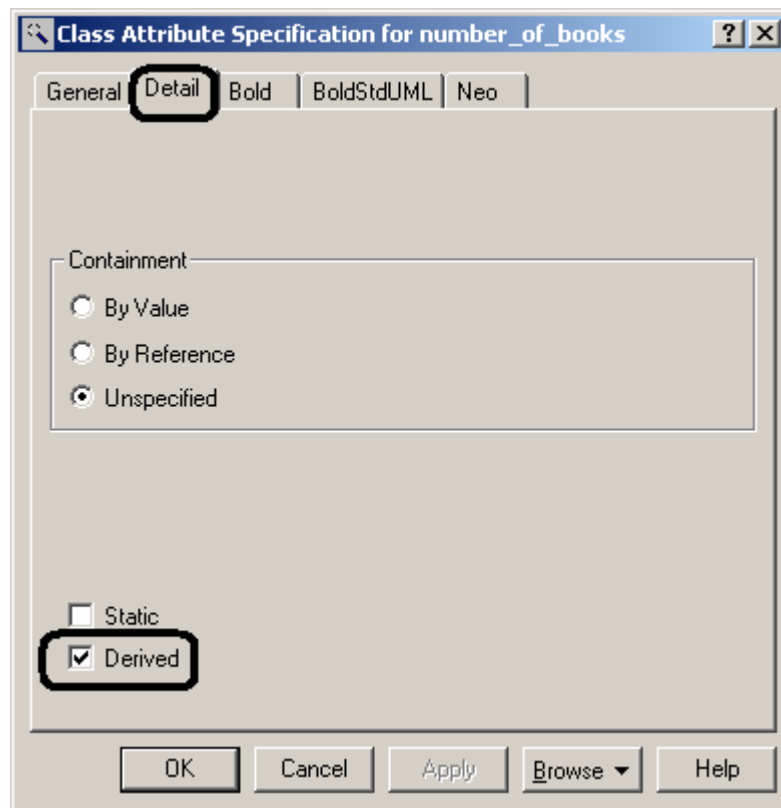


Fig. 5.2. Adjustment of derived attribute character

After double click on the given string the editing window will be open for entering OCL-expression. For “number_of_books” considered attribute we shall enter the following OCL-expression: books-> size, i.e. total of books. After closing a window of the specification and saving UML-model the information on derived attribute will be saved in model.

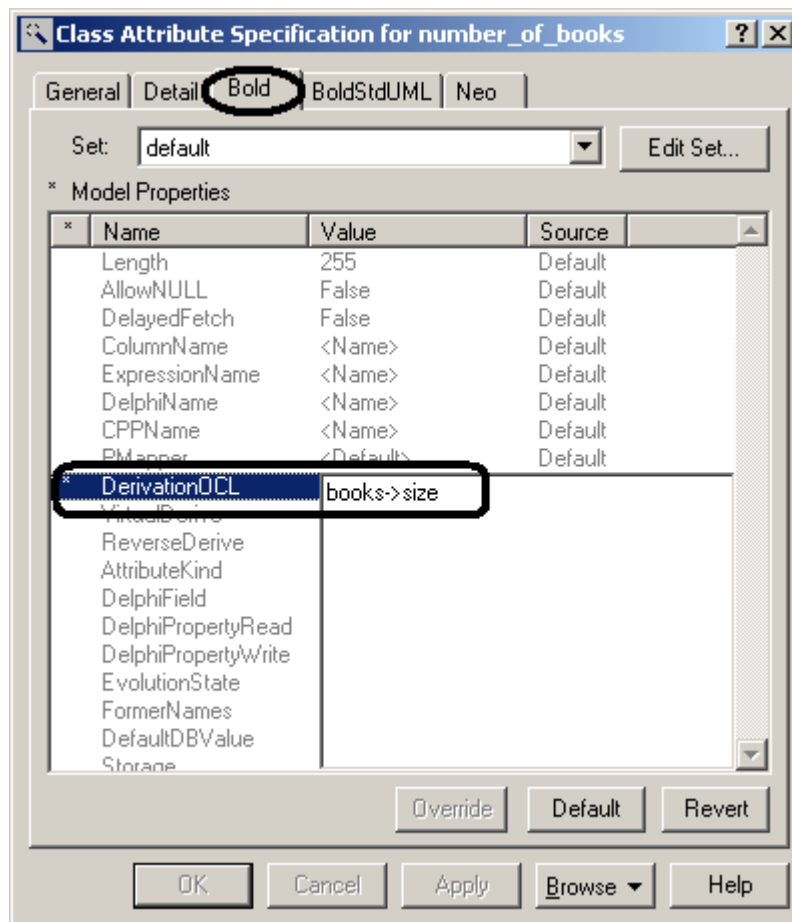


Fig. 5.3. Specification of OCL-expression for derived attribute

Forming Derived Attributes in the Built-in Editor of Models

Also it is possible to set derived attributes in the built-in editor of models in Bold for Delphi environment. For this purpose it is necessary to select the necessary attribute in model tree, then make active "Derived" tag on the right of the properties panel (see Fig. 5.4.), and enter OCL-expression into the window named "Derivation OCL" (see Fig. 5.4).

NOTE

For forming OCL-expressions in Bold for Delphi models editor it is possible (and it is more convenient) to use the built-in OCL-editor instead of "manual" input. It will be considered in Chapter 8.

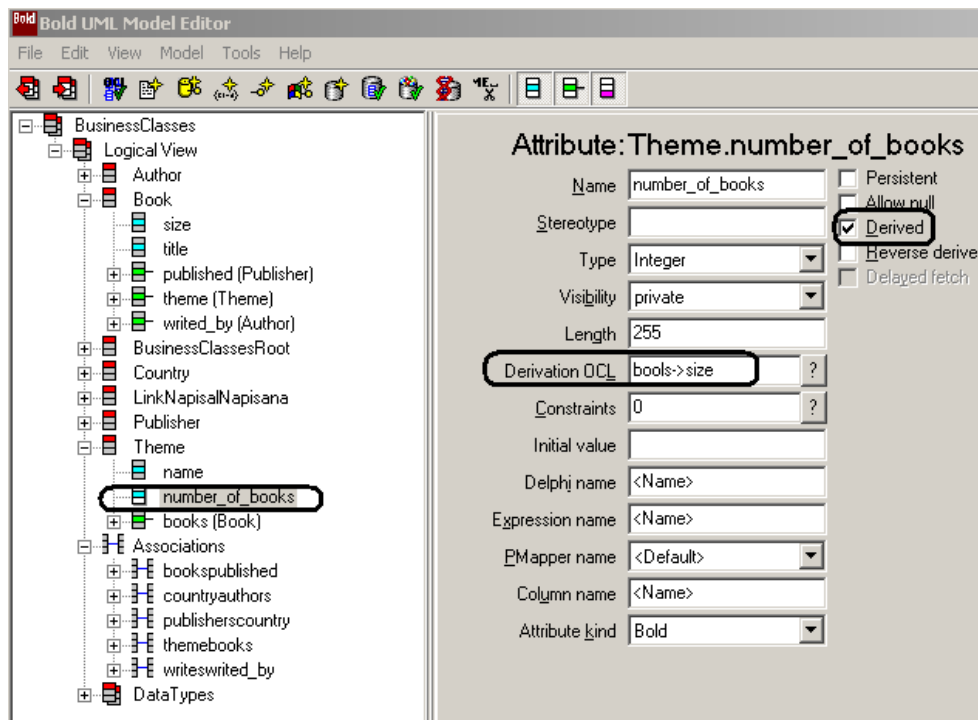


Fig. 5.4. Forming the derived attribute in Bold Model editor

Derived Associations

In Chapter 1 when we considered bases of UML the concept of association as kind of the relation between some classes has been described. At that examples of “usual” associations which roles are fixed and specified have been considered. In this section we shall get acquainted to *the derived associations* having some other character of model elements links. The derived association has no roles on the ends fixed beforehand, and instead of this such roles are set by special OCL-expressions, and their values “are derived” at application run.

NOTE

The description of derived associations demands understanding of OCL bases. Just by this reason the given concept was not considered earlier, in particular, in Chapter 1.

For the better understanding of derived associations we shall consider a small example.

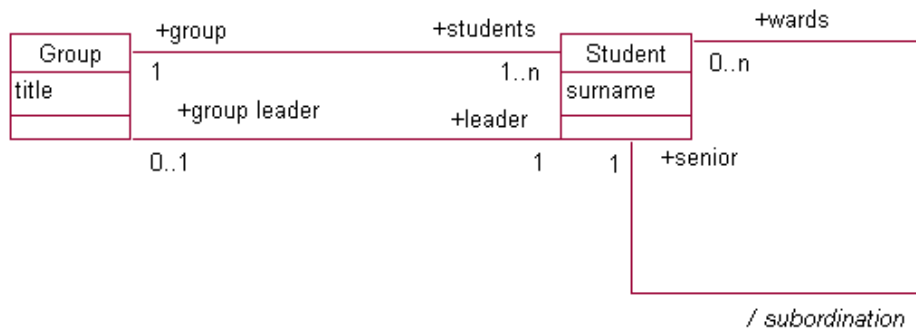


Fig. 5.5. Model with derived association

The model (see Fig. 5.5) includes two classes – “High school group” and “Student”, and it is described by the following basic business-rules:

- ❑ The group includes several students;
- ❑ Each student belongs only to one group;
- ❑ Each group has one leader from among students.

Besides in “Student” class there is a derived association with “subordination” name. We shall consider it more in detail. This association shows that each student has one senior student (“Senior” role of derived association). On the other hand, each student can either have no “wards” in general, or have several “wards” students (“Wards” role of derived association). From the point of view of a described subject domain it is possible to formulate the following obvious regulations:

1. For each student the senior is the head of group in which the given student studies (rule #1).
2. For the senior student wards are students of that group in which he is the leader (rule #2).

These regulations allow to generate OCL-expressions for roles of derived association as follows. From a rule #1 follows, that for “Senior” role it is possible to write the following expression in OCL – “group.leader” where “group” represents educational group in which the given student studies, and “leader” represents the student-leader of this group. It is necessary to emphasize, that here we again use effectively capabilities of navigation on model by means of OCL-expressions.

Similarly using a rule #2 it is possible to write OCL-expression for the second role of “Wards” derived association. It will look as follows: “group leader.students”, setting a collection of students of that educational group in which the senior student is the leader. Thus, derived associations allow connecting dynamically elements of model, proceeding from other associations already available. There is no doubt that it is convenient from the point of view of automation and increasing flexibility of development.

Formation of OCL-expressions for roles of derived associations is carried out similarly to the rules for formation of derived attributes described above. These operations can be made both in Rational Rose (see Fig. 5.6),

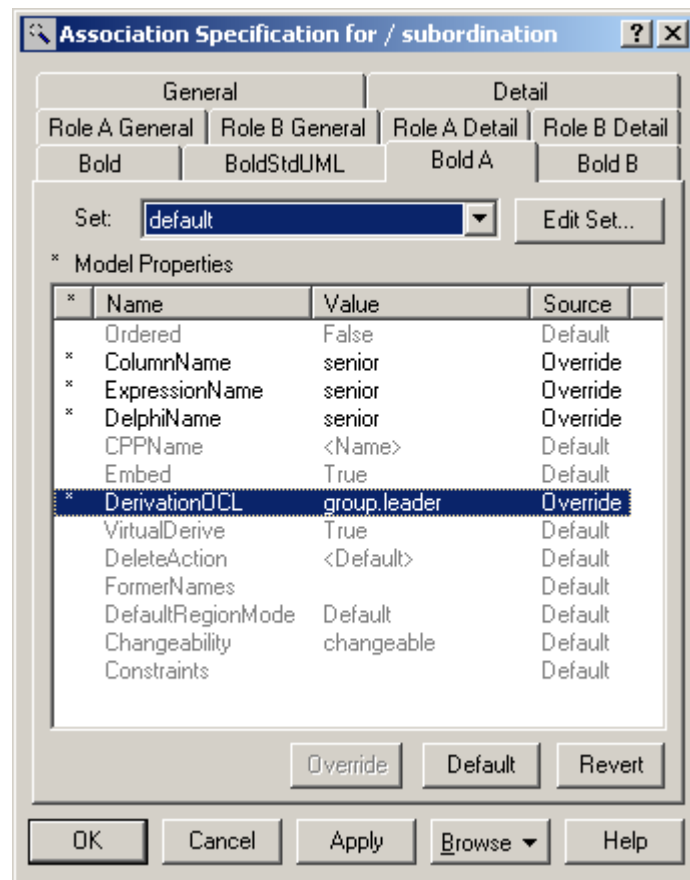


Fig. 5.6. Specifying “Senior” role in Rational Rose

and in built-in editor of models.

At that it is necessary to set preliminary corresponding “Derived” tags for specifying an attribute, that the given association is derived.

Forming Constraints

Also OCL is widely applied for creating conditions-restrictions for elements of model (actually, for this purpose OCL has been originally developed). In this case for specifying each constraint at model layer OCL-expressions are also used. For example, for Author class we can write the following constraint `<surname <>`, i.e. to forbid inclusions of the new author with “empty” string-surname in structure of authors.

Forming OCL-constraints in Rational Rose

For forming constraints in Rational Rose it is necessary to turn to the window of the class specification (we shall remind, that it is called by double click on the class image or from the pop-up menu). After that on “Bold” bookmark it is necessary to select “Constraints” tagged value (see Fig. 5.8), and, after double click on this string to write in editing window necessary OCL-expressions. In this case for Book class we have imposed “size>50” constraint (see Fig. 5.8) that means constraint on low value of book pages number equal to 50 pages (we shall remind, in order to prevent probable confusion that in this case “size” is a name of attribute in a class instead of OCL operator).

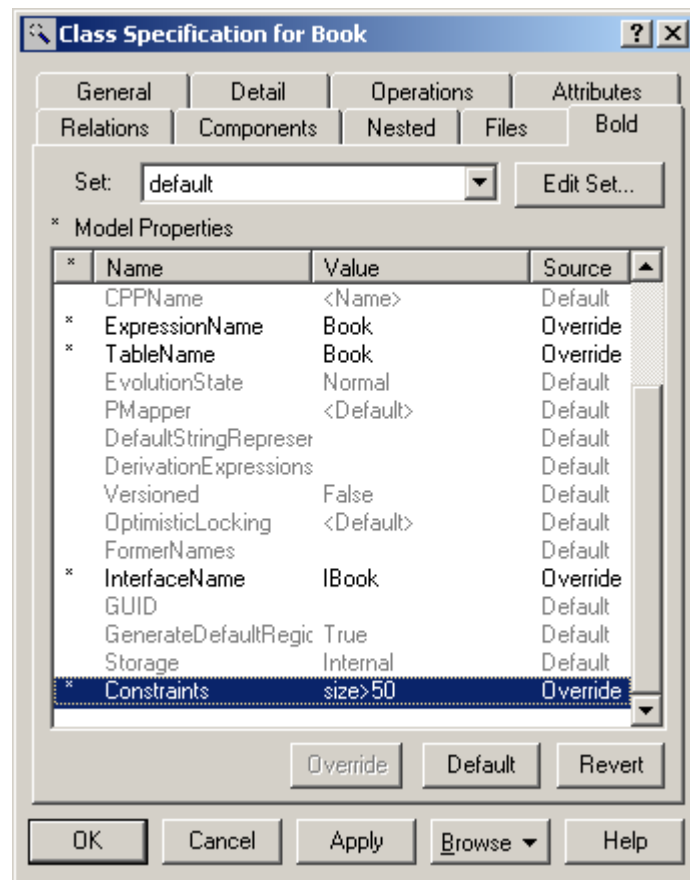


Fig. 5.8. Specifying constraint in Rational Rose

Forming Constraints in Bold for Delphi Built-in Editor of Models

For creating restrictions in the built-in editor the special editing window is used (see Fig. 5.6), in which the total of restrictions at the current element of model is screened. For forming restrictions it is necessary to select the necessary element in the model tree, then press the button with question mark, located to the right of "Constraints" window. In result the special editor of constraints will open. This editor is very simple in use. It contains two buttons - for addition and removal of constraints, and also "name" and "body" fields for input by the developer of constraint name and OCL-expression accordingly.

Let's select Book class in the model tree and call the constraints editor. We shall add one constraint, having named it PageAmount (amount of pages), and enter body OCL-expression «size>50» into the field .

Let's add one more constraint, we shall name it Title_not_null, for which we shall enter OCL-expression «title<>''» (see Fig. 5.11). Thus we “have forbidden” to the book to have the empty name.

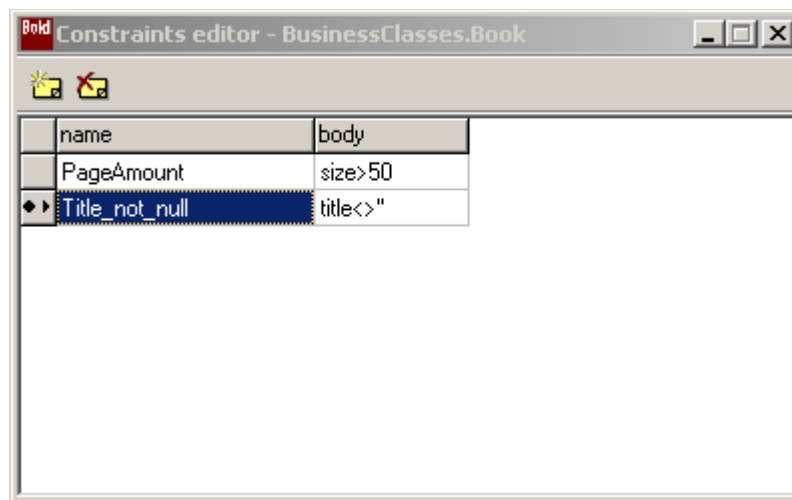


Fig. 5.11. Viewing editor with two constraints

Thus for any model element it is possible to set several constraints if necessary. At that situations can occur, when as a result of wrong or erroneous OCL-expressions input (for example, at OCL syntax failure) the entered expressions “will be inoperable” and will lead to errors of the application run. To avoid these situations in structure of the built-in editor means there is a tool for validation check of all OCL-expressions contained in model. For its call it is necessary to select Tools ► Validate OCL in model in the main menu of editor (see Fig. 5.12).

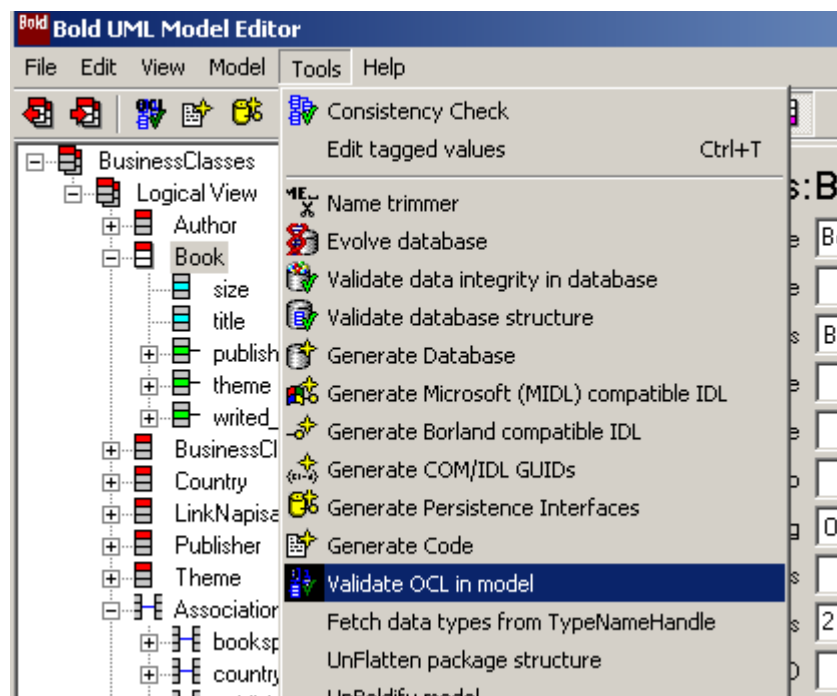


Fig. 5.12. Starting OCL-expressions validation check

NOTE

It is recommended to carry out such operation for validation check of the OCL-expressions used when forming derived attributes.

Features of OCL Dialect in Bold for Delphi Environment

Realization of OCL in considered Borland MDA (Bold for Delphi) version has some differences from OMG standard. In this section we list briefly the basic features of used OCL dialect and its differences from the specification.

- ❑ OCL standard allows to include in OCL-expressions the comments, designated by two dash symbols «--». OCL dialect used in Bold excludes use of comments.
- ❑ In structure of used OCL dialect «allInstancesAtTime (t-parameter)» operation presents allowing to get status of all object space objects for the specified moment of time used as parameter.

NOTE

For use of allInstancesAtTime operation it is necessary to install separately the obtained add-in module of Bold for Delphi environment, named «Object Versioning Extension» which supports “history” of object space changes in time.

- ❑ OCL standard uses “.size” operator for obtaining length of string attribute. OCL Bold-dialect allows to use for this purpose “.length” operator.
- ❑ For specifying “empty” initial value of attribute OCL Bold-dialect allows to use «NULL» expression.

OCL Dialect Extensibility in Bold

OCL dialect in Bold for Delphi environment provides developers with wide set of capabilities, due to which it is possible to solve practically all tasks. However, as a result of Bold practical implementation, developers in some cases face with necessity of extension of this set. For example, similar requirements arise when creating the “mixed” applications combining both Borland MDA tools, and elements of realizing the “traditional” approach of databases applications development. Apparently, taking into account this sort of situation, on the one hand, and also with the purpose of increasing convenience of forming OCL-expressions, on the other hand, some companies under own initiative (and completely free-of-charge, at least at the moment of this book writing) have developed and have begun to provide the supersets of OCL-operators for use in Bold for Delphi environment. Such sets of OCL-extensions are delivered as special program modules. Now there is an opportunity of free extension of OCL dialect, and amount of additional OCL operators in these sets is rather great - more than 80. More in detail the information on products of third-party companies is presented in Chapter 15.

Summary

In this chapter we have got acquainted with OCL – Objects Constraints Language. In spite of the name reflecting only one of OCL functions – the constraints set on model elements, it is obvious, that means of this language cover a much more spectrum of the capabilities given to the developer. Having the advanced means of work with collections, OCL gives very convenient and flexible capabilities of navigation on UML-model, which practically makes unnecessary use of SQL. OCL also possesses necessary means of work with metainformation, i.e. with the information on types of model elements. OCL syntax is simple and clear, convenient and transparent rules of OCL-expressions formation allow receiving easily readable, laconic and simultaneously powerful expressions playing a role of original «OCL-inquiries» to elements of object space. OCL role in formation of derived attributes and constraints set for UML-model elements is shown. The concept of derived associations, and formation of such associations roles by means of OCL-expressions is considered.

Certainly, in this chapter we have considered only base capabilities of language, as its full description would occupy much more place. But the information represented here is enough for understanding OCL importance, as one of basic MDA tools in general, and Borland MDA tool, in particular. In the subsequent chapters of the book, at the description of OCL practical use when developing MDA-applications, we shall refer time and again to the given chapter, which in this sense is some kind of the directory on OCL.

Chapter 6. Object Space

The concept of Object Space (OS) is the major element in Borland MDA architecture. Any developer using this technology, sooner or later inevitably faces with necessity of implementation of many capabilities provided by Borland MDA, however without understanding the bases of work with object space many of such capabilities remain inaccessible.

Object Space Concept

What does object space represent from the point of view of MDA-applications developer? On the one hand, it can be considered as the specific container, in which during the application run the objects representing realizations of UML-model elements are placed. From this point of view, the object space as a matter of fact is an instance of application model, on the analogy, as the object is an instance of a class in OOP. It means that in object space all entities and all links presented in model (classes, attributes, associations, roles) are implemented and filled with the concrete content. Thus, each object contained in object space contains the full information on the properties (attributes, operations, etc.) and about relationships-links with other objects.

On the other hand, the object space represents the program buffer (cache), which accumulates the changes occurring in system that arise, for example, at removal, change or addition of objects, including transaction operations realization. Thus special mechanisms keep track of such changes accumulation, and calculate difference between elements-objects state in memory and persistence layer state (of the database). When calling operation of persistence layer updating (UpdateDatabase) mechanisms of object space provide synchronization of memory and database contents. In this context the object space realizes analogue of known mechanisms of caching deferred updating CashedUpdates used at work with DBMS.

Besides the object space contains the special mechanism of mapping the data, named «persistence mapper», which serves as the peculiar channel between persistence layer (realized, for example, by means of DBMS) and the objects contained in Object Space. Thus, in Borland MDA-applications architecture the object space occupies the central part (see Fig. 6.1) realizing the following basic functions:

- ❑ Storage and representation of the information on realizing model elements during application run. Providing integral logic structure of the application. Processing of internal events of system.
- ❑ Tracking of changes occurred in system and forming delta-information on the cumulative differences with persistence layer;
- ❑ Interoperation with the graphic interface, providing the information on objects state;
- ❑ Interoperation with persistence layer, synchronization of memory and a database states.

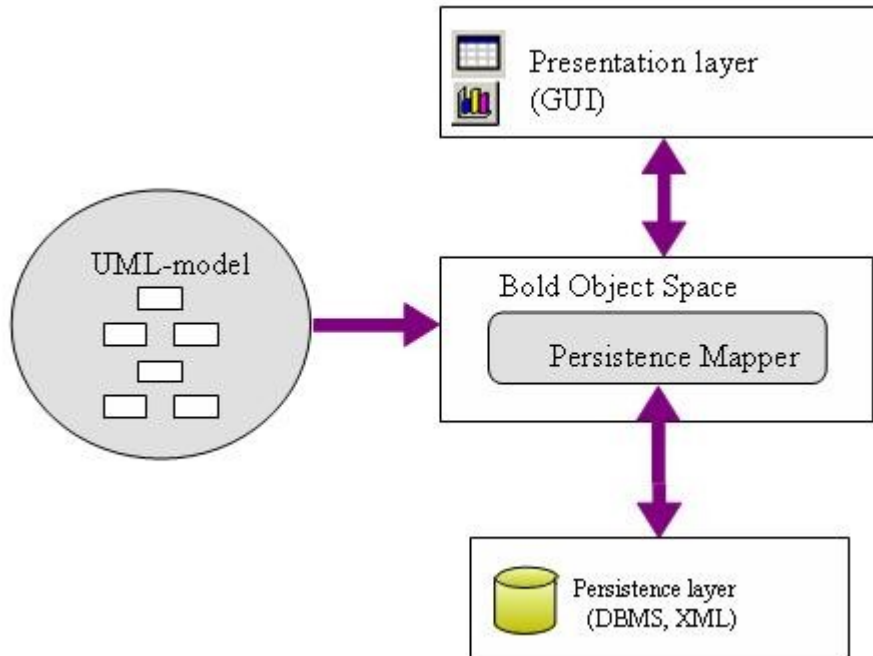


Fig. 6.1. Interoperation of Borland MDA components

Special classes model data carriers, and also classes realizing object space elements management and carrying out the system events processing are included for providing such functions in Borland MDA. For illustrating system tasks solution at this level we shall consider the problem of the control and management of objects life cycle during application run. When starting MDA-application the part of objects is loaded from persistence layer into object space (the developer at a stage of application creation or the user during application run can set necessity of loading one or another object). These questions will be covered in the subsequent chapters. At that the part of objects remains unloaded in memory, however, the system considers that they nevertheless exist, as they have not been removed during the last sessions. Thus, set of objects in memory and objects in DB is formed; such set forms the so-called “conceptual object space”. If an object removal takes place, it nevertheless remains in memory and in DB, until UpdateDatabase method is not called. Till the moment of this call the removed object is marked by system as removed, but it will continue to exist both in memory, and in DB, however, it will not exist in conceptual object space, and becomes inaccessible for use. For managing such situations Borland MDA has special means, so-called *State Machines*.

OS Structure

The structure of classes hierarchy, providing functioning of object space, is rather complex and volumetric, and it includes hundreds classes. We shall consider only a few most

important fragments. All classes can be divided into two basic categories: internal and external classes. Internal classes have TObject class as the parent, and external classes – TComponent one. The hierarchy fragment of internal classes is represented in Fig. 6.2.

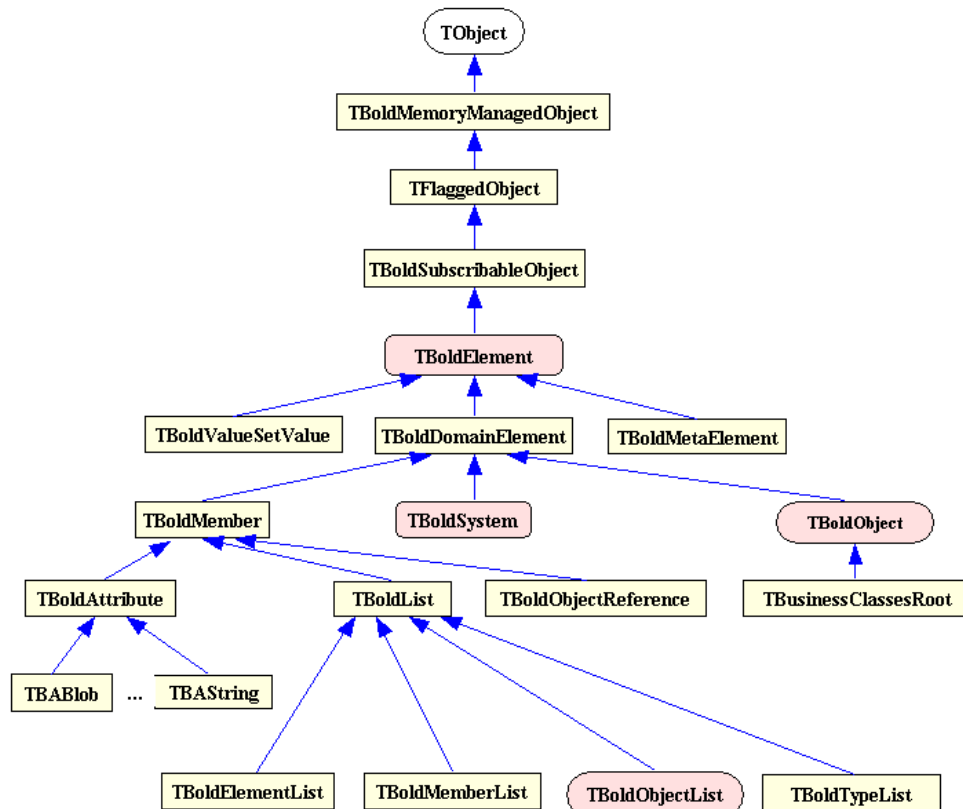


Fig. 6.2. The hierarchy of internal classes in Borland MDA

We shall describe briefly some of these classes.

TBoldMemoryManagedObject is the class-parent of this hierarchy of classes. Its occurrence is caused by the fact that Borland MDA has own manager of memory.

TBoldFlaggedObject class is intended for internal use, and contains a set of bit values-flags, and also a set of values of the listed type packed into an integer format. Flags basically are used for saving the information on various states of object space elements.

TBoldSubscribableObject class represents **the mechanism of subscribing to events** that is very important for Borland MDA. It will be considered later, now we shall only note, that this class allows realizing powerful and flexible support of reactions to the various events occurring in system. Practically any components of Borland MDA including visual

components can serve as sources of the certain events, to which it is possible “to subscribe”, and, at their coming, can automatically execute preset reactions-handlers of events. The subscription mechanism is effectively used by Borland MDA environment, in particular, when working with derived attributes and associations, but this mechanism is also accessible to the developer providing unique capabilities on managing the application work in the event-organized software system.

TBoldElement class is the major element in the represented hierarchy and the central link in the object space description, being a super class for all its elements. It is the universal class, capable to represent either value, either object, or a *metainformation* (the information on type). Accordingly, TBoldElement has three children for representation of the specified three kinds of represented elements - **TBoldValueSetValue** (representation of values), **TBoldDomainElement** (representation of objects), and **TBoldMetaObject** (representation of the information on types).

TBoldSystem class represents object space as a whole, i.e. possesses the information on all instances of the application model elements.

TBoldObject class represents object of Borland MDA object space. At code generation all user classes are TBoldObject children. All objects in object space also are children of this class, or child relatively to its children.

All members included in any Borland MDA object (for example, attributes) belong to **TBoldMember** class. However, this class serves for other purposes too, and it has independent value, generating, for example, such class as TBoldList, which in turn is the parent of **TBoldObjectList** important class. As follows from the name, this class serves for representation of the objects list. At that to each model class an appropriate object-instance of TBoldObjectList class of the object space corresponds, and such object-instance contains an objects collection of a concrete model class.

TBoldAttribute class is also very important being a parental class (super class) for several classes, representing attributes of various types – string (TBAStrng), date (TBADate), BLOB bit array (TBABlob), etc. Thus, we see that Borland MDA has its own data types – analogues of standard Delphi-types. And it is not casual, as, Borland MDA represents some kind of “system in system” that is the independent program system functioning within the framework of certain operational system (Windows at present time), and possessing its own manager of memory, own advanced mechanism of generation and events processing and other similar properties. By request the developer can create and register new MDA data type, having taken advantage of the attributes creation master (to select Bold->Attribute Wizard... in Delphi menu).

Now we shall briefly consider hierarchy of external classes (see Fig. 6.3).

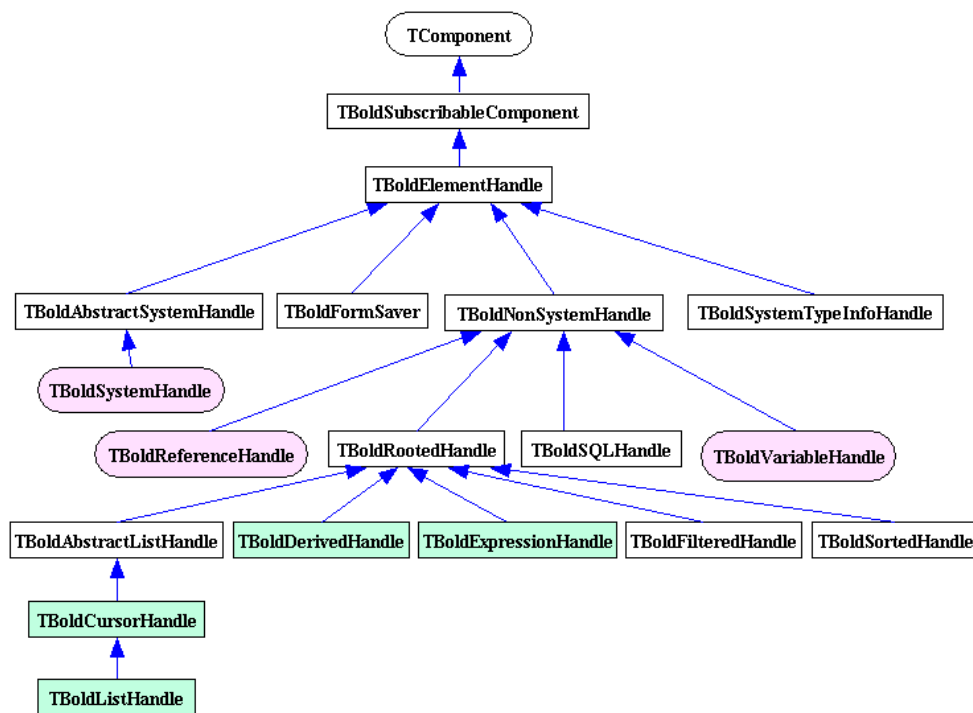


Fig. 6.3. The hierarchy of external classes in Borland MDA

All external classes have TComponent class as the parent. Naturally, it means, that we shall see the majority of these classes in a components palette of Delphi development environment. If internal classes can be considered as some kind of the “skeleton” supporting complex “organism” of object space, then external classes are more likely original “adapters” to the presentation layer and persistence layer. These classes are also the basic tools of the developer of applications in Borland MDA. The presentation layer and persistence layer we shall consider further in more details, and now we shall pay attention to the fact that the overwhelming majority of external classes contains in the name “Handle” term, i.e. *a descriptor*. The following chapter will be dedicated to handles and work with them.

TBoldElement Class

The given class, certainly, can be considered as the basic one in hierarchy of the internal classes forming object space. It is a super class for all elements of model, possessing ability to represent not only concrete value but also to represent objects and metainformation (the information on types). The basic methods included in TBoldElement class also are very important in all its child classes. We shall consider some significant properties and methods of this class below.

TBoldElement Properties

StringRepresentation[Representation:TBoldRepresentation]

Due to this property any model element in Bold for Delphi can be represented as a text string. Representation parameter defines a kind of string representation, which can be both brief, and verbose.

AsString

This property is equivalent to the previous one except that for string representation brDefault representation is used by default.

BoldType

This property returns the information on element type and at that it has TBoldTypeInfo type. Each instance of model element in object space always “knows” its own type during application run that is one of key features of Bold for Delphi environment.

Evaluator

This property represents “calculator” of expressions of TBoldEvaluator type connected with the given OS element. Each element has its own “calculator” used for estimating and obtaining results of OCL-expressions (for more information on OCL use together with methods of TBoldElement class see Chapter 8).

Mutable

Mutable logic property is responsible for capability of updating the given element. It can be used for check of capability of changing OS elements. Class object attributes can exemplify elements, for which this property usually has true value. On the contrary, for all elements representing metainformation this property always has false value, as during application run it is forbidden to change the model or its types. Any element can be made “unchangeable” by means of calling MakeImmutable method. However after that it is impossible to realize inverse operation, as Bold for Delphi environment keeps track such “unchangeable” elements occurrence, and, for example, with the purpose of run time optimization can place them in a special cache, copy into another memory space, etc, “at its own discretion”.

TBoldElement Methods

Assign(Source: TBoldElement); virtual Method

Assign method by analogy to known Delphi classes is used for copying OS source-element value into the current element. At copying check on identity of both elements types is carried out.

EvaluateExpression(...) Method

The given method is frequently applied at programming operations with OS elements. Its presence reflects one more key feature of Bold for Delphi – capability of each OS element to derive OCL-expressions and to return result as OS element. Use of this and similar methods will be considered in more detail in Chapter 8.

SubscribeToExpression(...) Method

This is one more important and key method of `TBoldElement` class. The sense of this method is “to subscribe” for changing OS element specified by OCL-expression-parameter. The mechanism of subscribing will be in more detail described in Chapter 14.

TBoldSystem Class

Here we shall more in detail get acquainted to some properties and methods of `TBoldSystem` class, as realization and management of object space as a whole is concentrated in it.

TBoldSystem Properties

DirtyObjects: Tlist Property

This property defines the list of all objects, which have been changed in memory, but have not been saved in a database yet. At attempt of terminating application with a nonempty state of the specified list, the corresponding exception will be produced.

PersistenceController: TBoldPersistenceController Property

It defines the controller of persistence layer for object space. If the specified property is empty, all object space is transient that is its state will not be saved in persistence layer.

Classes[index:Integer]: TBoldObjectList Property

It defines the list of all class objects with the specified index. All model classes are numbered starting with 0.

ClassByExpressionName[constExpressionName:string]: TBoldObjectList Property

It defines the list of all class objects with the specified name.

TBoldSystem Methods

MakeDefault Method

The application can include several object spaces, one of which is OS by default. The given method applied to a concrete instance gives “by default” status to the current object space.

StartTransaction(Mode: TBoldSystemTransactionMode = stmNormal) Method

It initializes sequence of transaction operations. The mode of transaction is determined by Mode parameter. Except for value by default (stmNormal) it is possible to use stmUnSafe value. Thus speed of data processing will increase due to exception of operations on objects and associations states checks; however the probability of functioning violation rises. Transactions can be enclosed in other transactions. In this case external transaction will be considered completed only in case of all enclosed transactions completion. For transaction termination CommitTransaction method is used (see further in this section), and for changes rollback RollBackTransaction method is applied.

CommitTransaction(Mode: TBoldSystemTransactionMode = stmNormal) Method

It completes the transaction started by mean of StartTransaction method. If for any reason transaction cannot be completed, exception is developed. There are two probable reasons when transaction completion is impossible. First, this situation occurs, if the given transaction includes the enclosed transactions, when even one of the enclosed transactions has not been completed, i.e. rollback of changes has been made. Second, there are special virtual MayCommit methods and bqMayCommit events connected to them, which are called when changing any element participating in transaction. If even one of calls of these methods returns False, then at attempt of transaction termination exception also will be produced. For easy processing of such situations it is possible to take advantage of special TryCommitTransaction method, which returns True, if the given transaction can be completed, or False if completion is impossible. In the latter case the given method provides automatic rollback of changes (RollBackTransaction is called).

UpdateDatabase Method

It makes OS synchronization with database level. All objects forming DirtyObjects list considered above, i.e. added, removed or changed from the moment of the last call of the given method, are saved in database or XML document.

Working with Object Space

In order Borland MDA classes hierarchy described above not to look as just abstract scheme being far from reality, it is useful to consider on concrete examples as work with these classes takes place in practice. In all examples of this chapter methods of working with OS, which do not require OCL use are shown. The advanced techniques of work with internal and external Bold for Delphi classes demanding OCL application will be considered in Chapter 8. As the application-basis for illustrating methods of work with OS elements we shall use the simple application created earlier describing the library catalogue (see Chapter 3).

Program management of OS objects attributes

NOTE

Bold for Delphi environment supports development of applications in two modes: without using code-generator, and with generating code of model classes. Features of work with code generation will be considered in Chapter 12. Working methods, without using code generation, are described in all other chapters including this one.

Let's set the following task in program way: to obtain a surname of the first author in the catalogue during application run. For achievement of this purpose **TBoldObjectList** class providing access to the concrete class objects collection is the best. Each instance of **TBoldObjectList** class represents the list of objects (instances) of the certain class of application model. Here it is possible to carry out some rough analogy to a database - that is it is possible to consider the list of objects of **TBoldObjectList** class, consisting of objects (actually of objects pointers), as the DB table consisting of records. Continuing analogy, we can conditionally assume that each object in the list has a set of attributes just as each record in the DB table has a set of fields. We shall start the decision of this task. As a class-source of the information we shall use a class with the name (Expression Name) "Author". For access to the list of all objects of "Author" class the following expression is used:

```
BoldSystemHandle1.System.ClassByExpressionName['Author'];
```

The following classes and their attributes are used in this expression:

- ❑ **BoldSystemHandle1** – Borland MDA component responsible for management by object space as a whole;
- ❑ **System** – property of **BoldSystemHandle1** component of **TBoldSystem** type (see Fig. 6.2);
- ❑ **ClassByExpressionName** – method of **TboldSystem** class, returning an expression of **TboldObjectList** type, in the given case it is the list of all objects-instances of "Author" class of our model.

TBoldObjectList class has **BoldObjects[i]** property of **TBoldObject** type, specifying an object in the list with *i* order number (at that numbering of objects in the list begins with 0). In turn, **TBoldObject** class possesses **BoldMemberByExpressionName ['member_name']** property allowing getting access to the specified attribute (a class member). Now we can generate the complete expression-operator for solving our task:

```
BoldSystemHandle1.System.ClassByExpressionName['author'].BoldObjects[0].BoldMemberByExpressionName['aname'].AsString;
```

For check of availability of the received operator we shall add on the form one **Button1** and usual **Label1**, and then we shall write the following event handler by keystroke:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption:=BoldSystemHandle1.System.ClassByExpressionName['author']
    .BoldObjects[0].BoldMemberByExpressionName['aname']
    .AsString;
end;

```

Let's add several authors using the Bold-navigator (see Chapter 3) and, after pressing the button, we shall make sure that our operator solves the set problem (see Fig. 6.4).

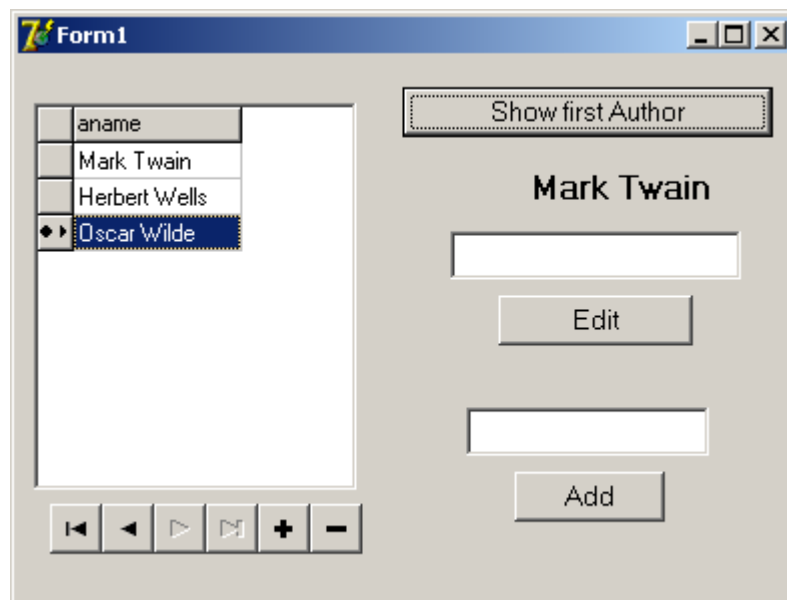


Fig. 6.4. Program mapping of object attribute

Now we shall complicate a task - we shall try to change in program way a surname of the first author. For this purpose we shall add on the form Edit1 editing field and second Button2 with the name "To change". We shall write the following keystroke handler for the second button:

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  BoldSystemHandle1.System.ClassByExpressionName['author']
    .BoldObjects[0].BoldMemberByExpressionName['aname']
    .AsString:=Edit1.Text;
end;

```

It is easy to see that we use the same expression for editing, but in this case we give to it a string of the text from editing window. After starting the application we shall enter

“Bernard Shaw” name into Edit1 field and after pressing the new button we shall make sure that the first author has changed. After pressing the first button we also make sure that from object space we obtain author already changed (see Fig. 6.5).

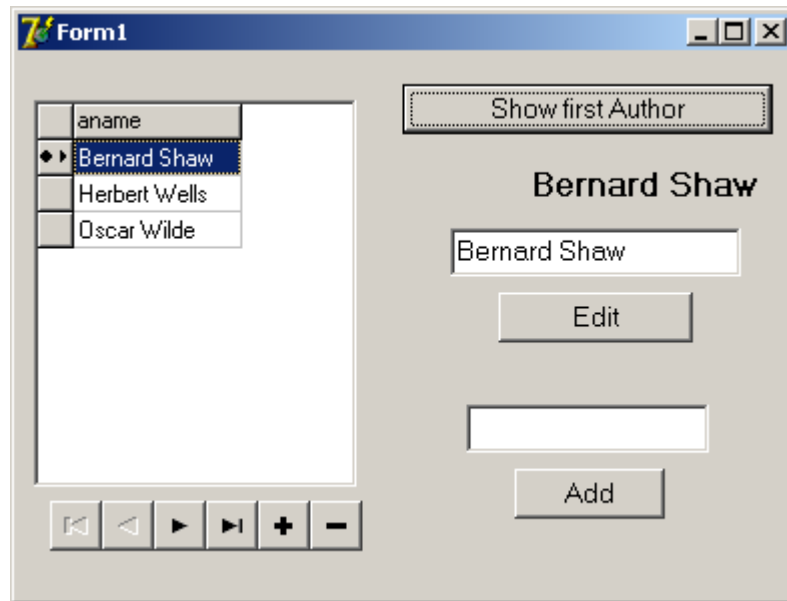


Fig. 6.5. Program changing object attribute

It is necessary to pay attention to that fact, that we did not use application code generation, and for this reason we have no the description of model classes at a level of program code, however, nevertheless, we have got name access to necessary attribute of concrete object during application run. It once again illustrates that fact, that the object space is a model instance and during application run it possesses the information on all its elements. On the basis of the simple mentioned example we can solve more complex problems as we, in point of fact, have already got now an opportunity to manipulate with attributes of objects-elements of authors collection.

Program Management of OS Objects

It is possible to go further, and, for example, to try to add in program way authors in the collection. For this purpose we shall use InsertNew (i:integer) method of TBoldObjectList class, where i is an order number of the collection element (we shall remind, that numbering is made from 0). We shall set the following task: by pressing the button to add the new author in the end of the list and to give to him the name entered by the user. For this purpose we shall place on the form one more editing window - Edit2 - for input of the added author surname, and the third Button3 with an inscription: “To add”. We shall write the following keystroke handler for the new button:

Listing 6.1. Addition of a new element in authors collection

```
procedure TForm1.Button3Click(Sender: TObject);
```

```

var author_count : integer;
begin
author_count:=boldsystemhandle1.
System.ClassByExpressionName['author'].Count;
boldsystemhandle1.System.
ClassByExpressionName['author'].InsertNew(author_count);
BoldSystemHandle1.System.ClassByExpressionName['author']
.BoldObjects[author_count].BoldMemberByExpressionName['aname']
.AsString:=Edit2.Text;
end;

```

Let's consider the code mentioned above. To place the new author on last place in the collection, it is necessary to obtain the information on number of its elements. For this purpose `author_count` integer variable and `Count` property of `TboldObjectList` class returning number of the list elements are used. Further the call of `InsertNew` method with `author_count` parameter is made, and due to that the new element is created at the end of the list. And, at last, as well as in the previous case, contents of `Edit2` editing field is given to an element of authors collection, however, not to its first element with 0 index, but to the last, with `author_count` index. Having started our application, it is easy to make sure (see Fig. 6.6), that new “Add” button now with success replaces the functional button of the Bold-navigator responsible for addition of objects, and besides provides transfer of the new author name from `Edit2` editing window into the list of authors.

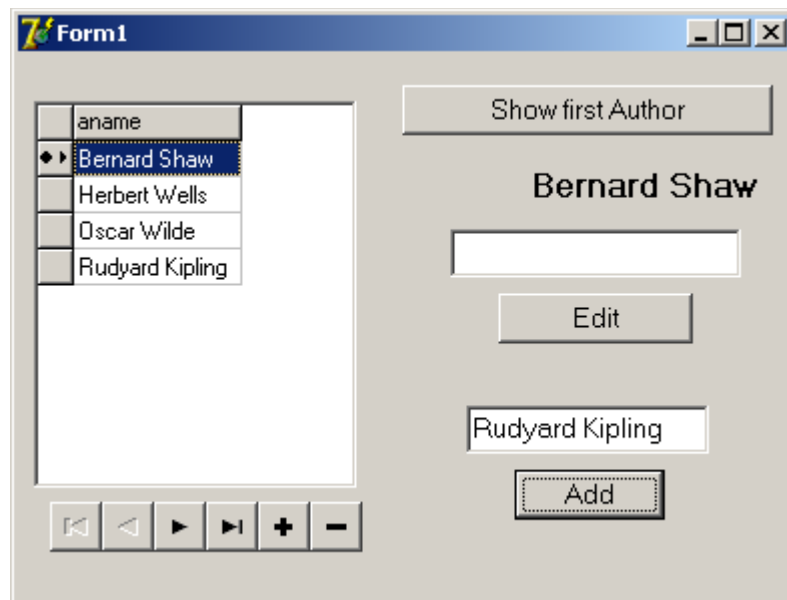


Fig. 6.6. Program addition of object

The considered examples can be modified easily and issued as procedures which further can be used for the different purposes, for example, for automatic program filling a database with any information, or replacement of functions of BoldNavigator standard component with own or third party components, etc.

Summary

In this chapter we have got acquainted with bases of constructing object space in Bold for Delphi environment. The object space is based on sets of classes which can be divided conditionally into internal and external classes (components). The basic properties and methods of two base internal classes - `BoldTboldElement` and `TboldSystem` - are described. Also working methods with objects of classes and their attributes from program code, without involving specific Bold-components are shown. Such methods of work are based on capabilities of `TBoldObjectList` class. The considered simple examples of a program code are called to help the developers-beginners “not to be afraid” to work at a program level with Bold for Delphi tools; besides these examples show, that, in spite of a plenty of classes Bold for Delphi program system is based on rather small “root” set of basic elements, to which the classes considered in the given chapter also concern.

Chapter 7. Handles

Handles Role

In practice when developing MDA-applications in Delphi 7 environment handles occupy, perhaps, the central place. Whatever functions the developer realize, be it the list-form information output from objects, sorting or filtration of the data, special queries to the DBMS and so on - all these functions are realized by means of handles. When in Chapter 3 we developed the simple MDA-application, we also used `TBoldListHandle` lists handles. All handles represented in Bold for Delphi components palette are not visual components. Their basic function is to provide various information links between object space, on the one hand, and the graphic interface and persistence layer, on the other hand. Handles form the majority from the basic elements of internal classes structure considered in the previous chapter, and all of them are children of `TBoldElementHandle` class, which, in turn, descend from `TComponent` class.

Classifying Handles

BMDA handles are subdivided into two basic types - *root* and *rooted*. Root handles, as it is easy to guess by their name, are primary sources of the information on object space (OS), while any rooted handle necessarily possesses "RootHandle" property. But the root handle not necessarily should act as value of this property, in other words rooted handles can be united in chains, at that the mentioned "RootHandle" property of the subsequent member of a chain shows the previous member. However for the first member of such chain this property should specify a root handle.

By functional purpose all handles can be divided into the following basic categories:

- ❑ OS handles; provide interoperability with object space;
- ❑ Persistence layer handles; provide functioning the interface with DBMS and XML;
- ❑ Handles of remote OS; provide functioning the distributed multilink applications.

In this chapter we shall concentrate the attention on handles of object space, which are represented on <BoldHandles> bookmark of Delphi components palette. The following components concern to them.

Root Handles:

- ❑ `BoldSystemHandle` – represents object space integrally;
- ❑ `BoldReferenceHandle` – represents the reference to the OS object;

- ❑ **BoldVariableHandle** – provides information storage on OS variable;

Rooted Handles:

- ❑ **BoldSystemTypeInfoHandle** – represents the information on UML-model;
- ❑ **BoldExpressionHandle** – represents result of the OCL-query applied to a root handle;
- ❑ **BoldDerivedHandle** – represents result of information processing from a root handle;
- ❑ **BoldListHandle** – returning the current element represents the list of objects;
- ❑ **BoldSQLHandle** – provides realization of SQL-queries to the persistence layer;
- ❑ **BoldCursorHandle** – transforms value into the elements collection;
- ❑ **BoldOCLVariables** – the auxiliary component providing “implementation” of OS-variables in OCL-context;
- ❑ **BoldUnloaderHandle** – provides an automatic roll-out of unused objects from OS.

Further in this chapter we shall consider all components listed above, except for **BoldSQLHandle**, which we shall describe in Chapter 10 dedicated to the work with a persistence layer. The sequence of the further description of OS components-handles corresponds to the order of their arrangement in Delphi components palette.

BoldSystemHandle

This component presents in each application created with the help of Borland MDA for Delphi 7. It actually represents in IDE (Delphi Integrated Development Environment) **TBoldSystem** class described in the previous chapter (see Chapter 6). Thus the given component is responsible for the object space integrally. External interfaces of the component are represented by **SystemTypeInfoHandle** and **PersistenceHandle** properties-components (see Fig. 7.1). At that values of these properties are references to the corresponding BMDA components. The component specified in **SystemTypeInfoHandle** property provides the information on types of UML-model elements. The component specified in **PersistenceHandle** property provides link with a persistence layer. If such component is not specified, OS is completely transient, and will not keep its state between sessions. Thus, **BoldSystemHandle** possessing the information on model at a stage of the application run provides OS interoperability with a persistence layer (DBMS), in that way taking the central position in the organization of functioning object space integrally. The application can use several components of this kind, at that several OS are formed correspondingly. **BoldSystemHandle** component has the following properties:

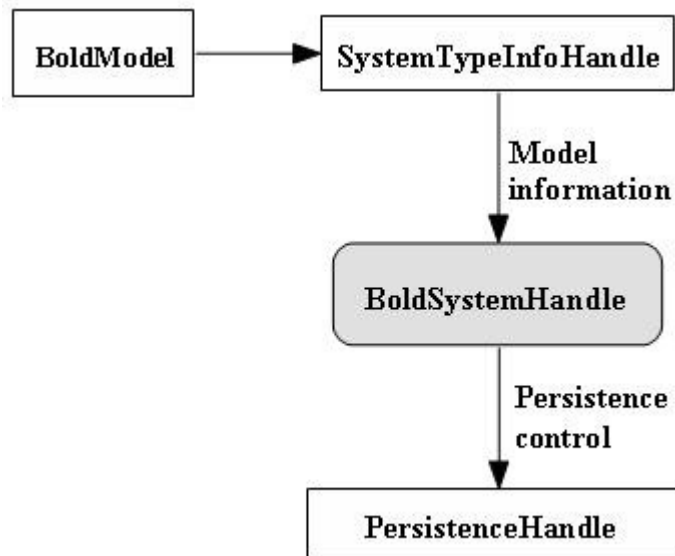


Fig. 7.1. Structure of information links of BoldSystemHandle component

- ❑ isDefault (logic) – defines whether the given component represents OS with the status “by default”;
- ❑ Active (logic) – sets a mode of OS activity or passivity; in particular, when OS activating there is an activation of a persistence layer, connection to the data and their loading; as a rule, at the stage of development this property is recommended to be set in False, and for automatic OS activation it is necessary to use the following property;
- ❑ AutoActivate (logic) – defines whether OS is activated automatically by the first request. For example, if any visual component is connected to the given OS, and the form containing such component is created, an attempt to activate OS will be undertaken, if the given property is set in True.

The given component, as a rule, is the basic source (RootHandle) for other OS rooted handles used in the application.

BoldSystemTypeInfoHandle

The given component-handle is intended for realization of one main function - to provide delivery of the information on UML-model to BoldSystemHandle considered in the previous section (see Fig. 7.1). The information on model includes basically a metainformation, i.e. the description of model elements types. At that the given component

obtains metainformation from BoldModel component representing application model, and transforms it to the special format optimized from the point of view of processing rate. The single external interface of a component is BoldModel property-component. Other properties are listed below:

- ❑ UseGeneratedCode (logic) – informs a component, whether the model classes code has been generated; metainformation on model depends on it. If code generation took place, names of model elements are equivalent to the generated names of classes (taking into account tagged values, see Chapter 4). If code generation was not carried out, all model objects will be formed as TObject and TObjectList types;
- ❑ CheckCodeChecksum – specifies to the component, whether it is necessary when starting to compare the code checksum to its source value formed at code generation. Such check serves the purposes of additional provision of reliability. This property is ignored, if code generation of a was not realized.

BoldExpressionHandle

It is intended for processing the information from a root handle by means of OCL-expression. It can function in structure of handles chain. For an illustration of work with this component we will address for example, considered at creation of simple expression (see Chapter 3).

Let's place on the form BoldExpressionHandle1 component from <BoldHandles> bookmark and BoldLabel1 visual component from <BoldControls> bookmark (see Fig. 7.2).

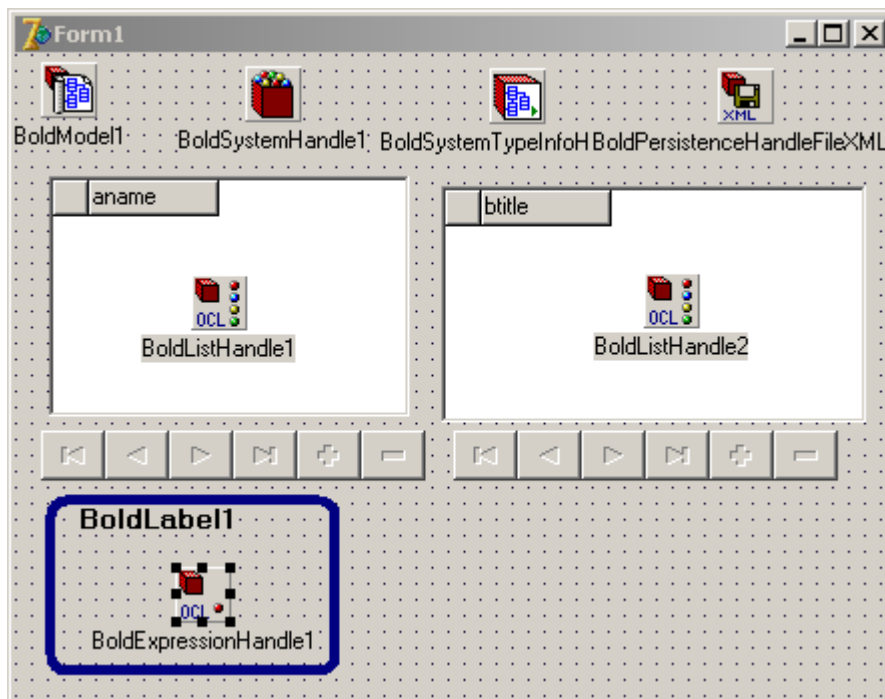


Fig. 7.2. Viewing form with new components

We shall pose the following problem - to map on the form total number of authors by means of BoldLabel1. For solving this problem we shall connect BoldExpressionHandle1 component to the root handle, having set RootHandle property in BoldSystemHandle1 value (see Fig. 7.3), i.e. thus we have specified to a new component-handle that the system handle will be a source of information for it. We shall introduce the following OCL-expression

```
Author.allInstances->size.asString
```

into Expression property of our component (see Fig. 7.3). This expression, as we have already known (see Chapter 5), returns the total number of authors transformed to the string type.

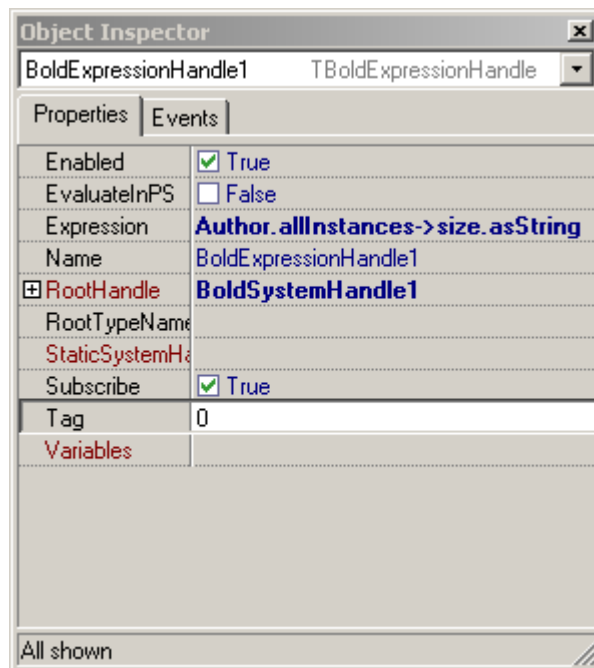


Fig. 7.3. Setting of BoldExpressionHandle

In order that BoldLabel1 maps the necessary information, we shall connect it to our handle by setting its BoldHandle property in BoldExpressionHandle1 value (see Fig. 7.4).

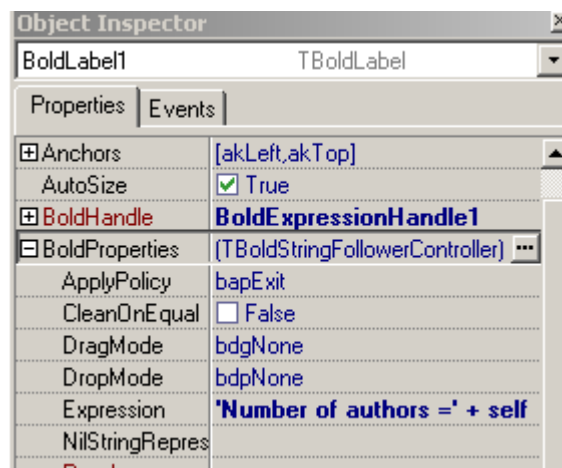


Fig. 7.4. Setting of BoldLabel properties

Further, in Expression subproperty of BoldProperties we shall enter the following OCL-expression:

```
'number of authors='+self
```

In this case the result will be the sum of two strings, last of which – *self* – returns a context of OCL-expression (see Chapter 3), and since BoldExpressionHandle1 is chosen as a source of the information, then instead of *self* the result returned by expression set for this handle will be substituted, i.e. number of authors.

Besides it is possible to adjust color and a font in usual way. After starting the application we shall make sure that the task is solved (see Fig. 7.5).

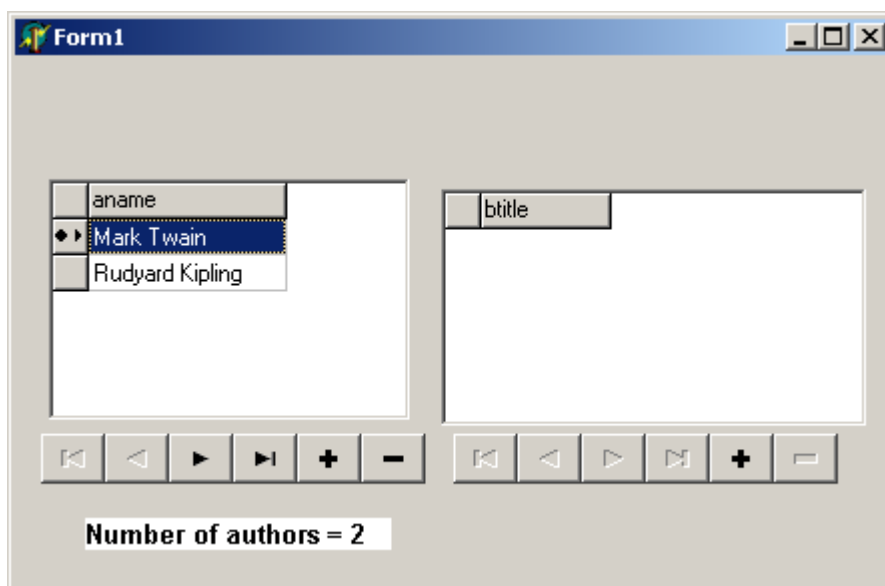


Fig. 7.5. Viewing the application in work

Moreover, when adding or deleting authors during work with the help of the navigator, it is easy to notice that our label “traces” all changes mapping the correct information.

There can be a natural question: why cannot we manage without a new handle of BoldExpressionHandle expression, just having connected our label to BoldListHandle1 handle of authors list, which has been already available? The answer is simple: specificity of BoldListHandle descriptor of the list is such that it always returns as a context only one the current element of the list instead of the whole list. Therefore «-> size» OCL-operation in this case will always return value 1, instead of number of the list elements (authors). However in the described simple example there is nevertheless an opportunity to do without a new handle. The fact is that our BoldLabel1 can be connected simply directly to BoldSystemHandle1 system handle; and OCL-expression for a label can be written down in the following form:

```
'Number of authors='+Author.allInstances->size.asString
```


It is easy to check up that in this case the posed problem is also solved. But such simple situation as in our example is quite rare. In the following chapter dedicated to OCL using we shall consider more complex example, and we shall show additionally examples of using a handle of expression. Now we shall only note that direct connection of visual elements (labels, etc.) to the system handle is not quite good practice. Intuitively we understand that it is not necessary to use the system handle representing object space integrally for such simple problems as labels generation. For these purposes other handles are intended, which allows in addition to structure the application, having entered the certain levels of information representation by a deductive principle.

Other properties of a described component concern to mechanisms of interaction with a persistence layer and to mechanisms of subscribing, which are common for the majority of OS handles, and will be considered in the following chapters.

BoldDerivedHandle

This component also is intended for the decision of problems of the information processing received from a root handle. Its difference from BoldExpressionHandle expressions handle is in the fact that the information processing is carried out without using OCL. In spite of capacity and flexibility of OCL language, in practice, though it is rare, there are situations, when OCL capabilities are limited. Then it is necessary to use program processing in base language (Object Pascal in Delphi). Just for such rare cases BoldDerivedHandle information processing handle is intended.

For illustrating use of a handle we shall place on our form from the previous example BoldDerivedHandle1 component and one more label – BoldLabel2.

Let's adjust BoldDerivedHandle1 handle as follows (see Fig. 7.6):

- ❑ We shall give BoldListHandle1 (list of authors) to RootHandle property;
- ❑ For setting ValueTypeName property we shall call the selector of types by pressing the button with dots, and in the window of the selector we shall select consistently: "Lists->Class Lists->Collection (Author)". Thus we have specified that the type of output result of our handle will represent a collection of authors

For BoldLabel2 we shall specify our new BoldDerivedHandle1 for BoldHandle, and we shall enter "aname" OCL-expression, i.e. a name of the author, into a window of Expression property.

Let's pose the following problem, senseless from the point of view of practice: let on BoldLabel2 the name of the author is mapped only in the case, when the author with "Ilf" surname is selected. For using the capabilities of BoldDerivedHandle it is necessary to write "OnDeriveAndSubscribe" event handler. We shall double-click on the objects inspector on the pulldown list near to the specified name of event, and then we shall enter a code according to Listing 7.1.

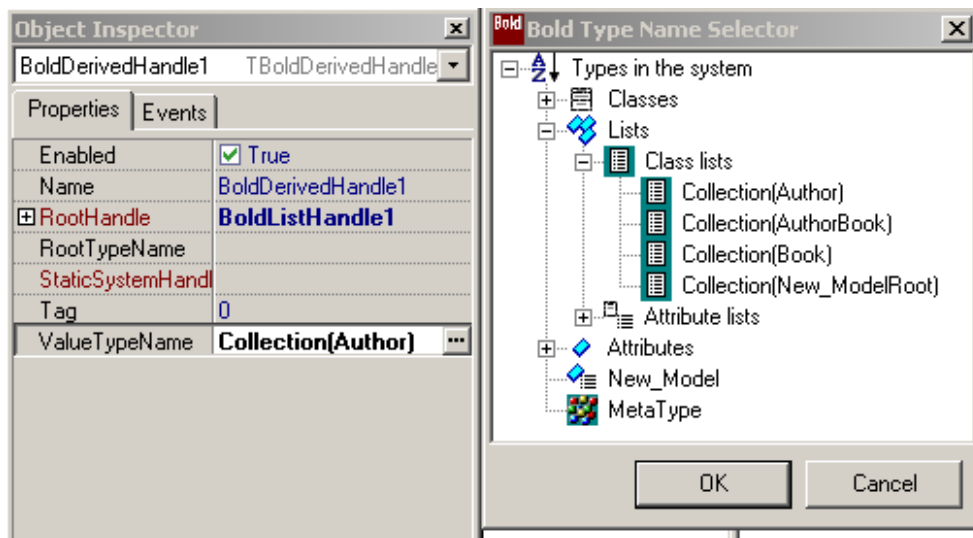


Fig. 7.6. Setting of BoldDerivedHandle properties

Listing 7.1. An example of procedure of event processing for BoldDerivedHandle

```

procedure TForm1.BoldDerivedHandle1DeriveAndSubscribe(Sender: TComponent;
  RootValue: TBoldElement; ResultElement: TBoldIndirectElement;
  Subscriber: TBoldSubscriber);
var st:string;
    E1 : TBoldElement;
begin
  E1:=BoldListHandle1.CurrentElement;
  st:=(E1 as TBoldObject).BoldMemberByExpressionName['aname'].AsString;
  if st = 'Ильф' then    ResultElement.SetReferenceValue(E1)
  else ResultElement.SetReferenceValue(nil);
end;

```

The output information of processing handle is transferred through ResultElement parameter. This parameter has TBoldIndirectElement type. This type is specially introduced for providing storage either the information or the reference to the object space element, which has been already present. As it is clearly from the given code example, for our simple case it is enough to check up a surname of the current author, having used the familiar expression:

```
BoldMemberByExpressionName['aname'].AsString
```

(see Chapter 6) for access to the class object attribute by name.

ATTENTION

It is necessary to understand clearly that this expression is not OCL-expression, but represents the usual fragment of a program code for the reference to `TBoldObject` class property.

Here, novelty is use of `E1` variable of `TBoldElement` type, which in this case is necessary for transfer as the reference at the output of the following procedure

```
ResultElement.SetReferenceValue(E1)
```

in case, when the current author is `Ilf`, or for transfer of the empty pointer otherwise:

```
ResultElement.SetReferenceValue(nil).
```

In this case the given example is a little bit artificial, and is intended only for the general illustration of work with `BoldDerivedHandle`. The careful reader will write OCL-expression for `BoldLabel2` without effort, solving the same task without using the described handle.

BoldVariableHandle and BoldOCLVariables

`BoldVariableHandle` variable component-handle is interesting from the point of view of the fact that it provides a capability of original “gateway”, through which “usual” Delphi variables can “penetrate” into object space and be used there for various operations along with “native” elements. Such problem can emerge frequently at practical development of MDA-applications. We shall remind that the object space by default includes only those elements, which are represented in UML-model. Let’s consider the following example. We shall assume that we want to carry out a filtration of authors depending on number of the written books. Let’s define it more concretely: when the user inputs any number in the window on the form, in the list of authors there should be only those authors, who have written such number of books, which concur with the entered number. The “mixture” of environments and means of programming is novelty in this problem. Let’s place `TEdit1` OCL-component on our form for entering number of books; `BoldOCLVariables1` component and `BoldVariableHandle1`, to which we shall give `bhNumBooks` name (see Fig. 7.7). `Edit1` component belongs to “usual” OCL-components while `BoldGrid1` grid belongs to MDA-components. The object space “knows nothing” either about `Edit1` component, or about values entered in its window.

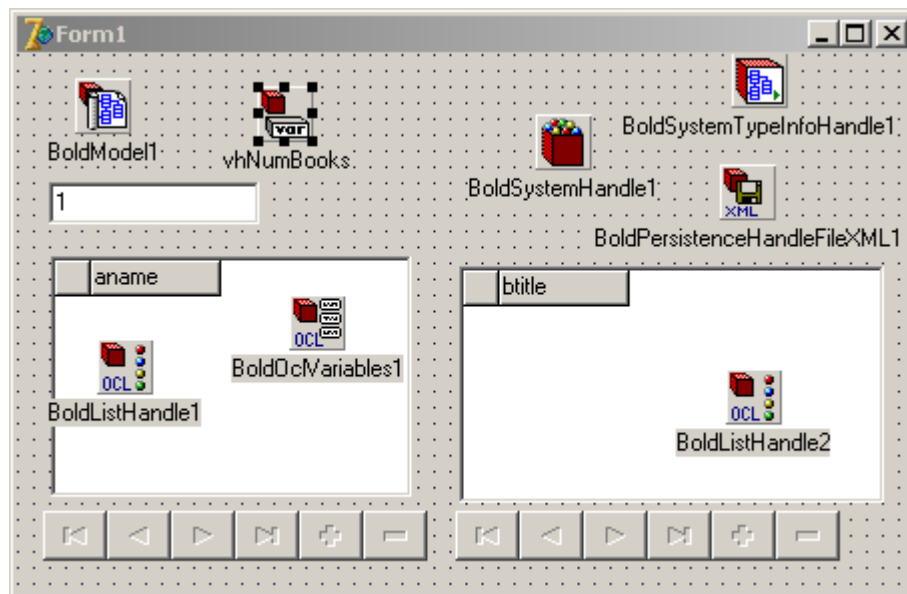


Fig. 7.7. Form for illustrating work with BoldVariableHandle

Just for organization of such link we use a variable handle. Let's adjust vhNumBooks component as follows (see Fig. 7.8):

- ❑ We shall give BoldSystemHandle1 value (from the pulldown list) to StaticSystemHandle property; StaticSystemHandle property specifies a source of the information on OS elements types;
- ❑ For setting ValueTypeName property value we shall open window of the types selector already known to us, and select Integer type; the given property defines type of the OS-variable, which in this case should be integer (number of books);
- ❑ We shall set initial value of our OS-variable, having opened InitialValues property, and having entered in the opened editor one string – "1".

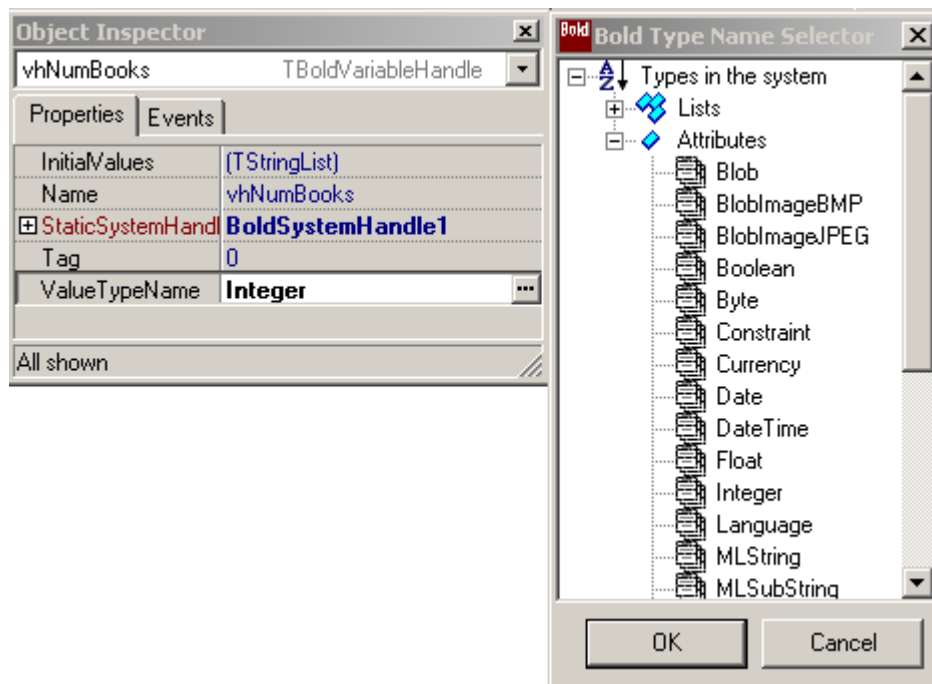


Fig. 7.8. Setting of BoldVariableHandle component

Let's pass to setting of BoldOCLVariables component. Having opened in the objects inspector Variables property of this component, we shall see the editor of variables set (see Fig. 7.9). It is constructed similarly to other Delphi editors. We shall add a new element in the variables list of the editor and adjust its properties. We shall set our vhNumBooks variables handle as BoldHandle, and we shall enter NumBooks name as VariableName. This name will be an alias of our OS-variable, and it is just the name that can be applied in OCL-expressions.

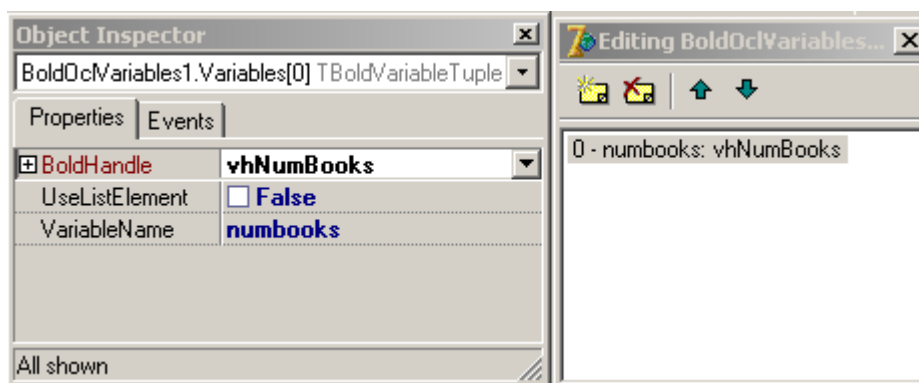


Fig. 7.9. Setting of BoldOCLVariables component

What for is BoldOCLVariables component intended? It represents the container-interface for several OS-variables (the variables available in the object space), and it becomes possible to use all variables, which are included in this container, not only in OS, but also in OCL expressions. And we shall do it just now. We will address to BoldListHandle1 authors list handle. First, we should specify to this component that besides model objects it also has an access to our OS-variables. For this purpose in the pulldown list of Variables property of this component we shall set BoldOCLVariables1 value. Second, having double click on Expression property, we shall open built-in OCL-editor and enter the following OCL-expression instead of available expression (see Fig. 7.10):

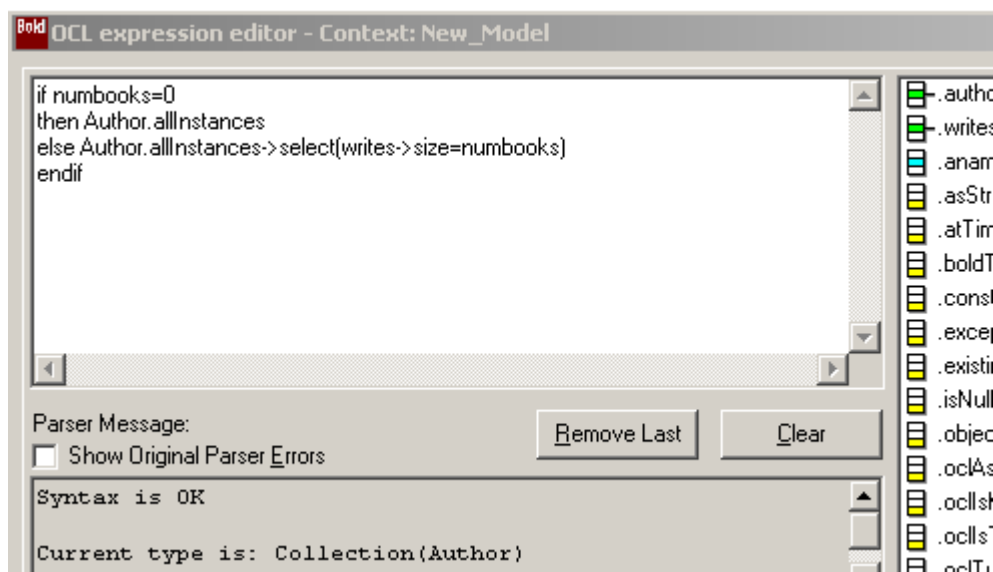


Fig. 7.10. Built-in OCL-editor

```
if numbooks=0
then Author.allInstances
else Author.allInstances->select(writes->size=numbooks)
endif
```

This OCL-expression contains the condition statement. If numbooks value of OS-variable equals to “0”, then the previous expression is used (i.e. all authors are selected), otherwise necessary authors fetching is carried out (OCL bases are stated in Chapter 5). What for is check on a zero used in OCL-expression? It is made for providing a possibility of editing the authors list. As we shall see soon, the filtered authors list does not allow changes.

So, we have adjusted a handle of OS-variable, and have built it in the container of OCL-variables. It is necessary to write the program code transferring contents of editing window

into the OS-variable. For this purpose we use OnChange event of Edit1 component, and we shall write the procedure represented in Listing 7.2.

Listing 7.2. An example of the value transfer in the OS-variable

```
procedure TForm1.Edit1Change(Sender: TObject);
var st:string;
begin
    st:=Edit1.Text;
    if st<>'' then vhNumBooks.Value.SetAsVariant(strtoint(st));
end;
```

From the given code fragment it is clear that the value of the number entered into Edit1 editor is assigned to Value property of our OS-variable.

ATTENTION

It is necessary to pay attention once again to the fact that in the text of the program vhNumBooks variable name is used while in the text of OCL-expression the alias of this variable – NumBooks is used, which has been set in BoldOCLVariables1 container.

Let's start the application run. As our OS-variable has “1” value by default, we shall receive already filtered list of the authors, each of whom has written one book. At that the navigator buttons responsible for editing the list of authors became unavailable.

For editing the list of authors we shall enter in the editor window “0” value, and after this it is possible to add new authors and new books in order to make sure on practice in correctness of functioning of the created application.

BoldListHandle

This component is used very frequently, basically as a handle for such visual components as BoldGrid or BoldListBox, i.e. there, where it is necessary to receive the list of elements. Examples of using BoldListHandle were already given on pages of this book time and again; therefore here we shall accent our attention only on features of its implementation.

First, it is necessary to repeat once again, that this handle differs from other considered handles in the property that as result it always returns a single value, instead of elements collection. This value is defined by the current internal indication of the list handle, which position can be operated in program way with the help of navigation methods given in Table 7.1.

Table 7.1. Navigation Methods of the List Handle

Method	Description
First	Moving the indicator on the first element of the list
Last	Moving the indicator on the last element of the list
Next	Moving the indicator on the next element of the list
Prior	Moving the indicator on the prior element of the list

It is easy to see that the specified methods are similar to the corresponding methods of DataSet component.

Besides the handle of the list possesses a number of specific properties, use of which lightens the work with elements collections. Some of these properties are given in Table 7.2.

Table 7.2. Properties of the List Handle

Property	Type	Description
Count	Integer	Number of the list elements
CurrentIndex	Integer	The current value of the indicator
HasNext	Boolean	Shows whether there is the next element
HasPrior	Boolean	Shows whether there is the prior element
CurrentElement	TBoldElement	Returns the current element
CurrentBoldObject	TBoldObject	Returns the current object

Besides BoldListHandle has the interesting property – MutableListExpression, which we shall examine more in detail. As we saw in the previous example, in the “filtered” state the list handle does not allow to edit the list itself, but it only allows navigation on it (see Fig. 7.11). Such state of the list occurs at imposing some constraint (filter) on the elements collection. To avoid such constraint the specified property - MutableListExpression is used. It represents the OCL-expression, which does not contain constraint. Editing structure of the list is possible in this case; however for viewing the list modified in such a way, it is necessary to use other special property - MutableList. Why does not the list allow the editing in the state of a filtration? It can be explained using the previous example. We shall assume that we add the new author in the list of authors. But directly at addition it is not known whether it satisfies to conditions of a filtration, i.e. how many books he has written. Therefore the system “does not know” whether to map him in the grid or not. From the point of view of OS integrity in this case it is better to forbid in general direct updating that occurs in practice.

BoldCursorHandle

This component is intended basically for transformation of single values into the elements collection for the further representation of this collection in visual components such as BoldGrid or BoldListBox. It can be applied effectively together with BoldVariableHandle OS-variables handles. For an illustration of the cursor handle using we shall modify our application, which we considered at the description of OS-variables handles. We shall delete their forms and add BoldCursorHandle1 and one more grid - BoldGrid3 (see Fig. 7.12).

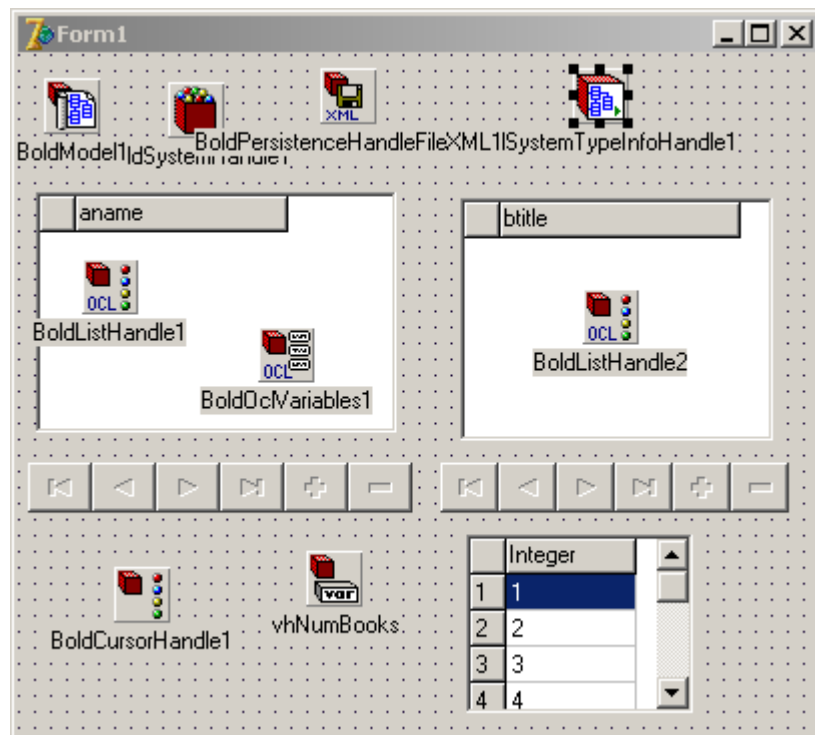


Fig. 7.12. The application form with the cursor handle

For this grid we shall specify new **BoldCursorHandle1** handle as **BoldHandle** property and set columns type by default. Further, we shall set **vhNumBooks** OS-variable handle created earlier as **BoldHandle** property for **BoldCursorHandle1** new component. We shall change a little properties of this OS-variable. First, we shall set not one value, as in the last case, but five strings with values from "1" up to "5" as initial values (**InitialValues** property) (see Fig. 7.13). Second, we shall change type of the OS-variable from **Integer** to the **Collection** (**Integer**) (see Fig. 7.14).

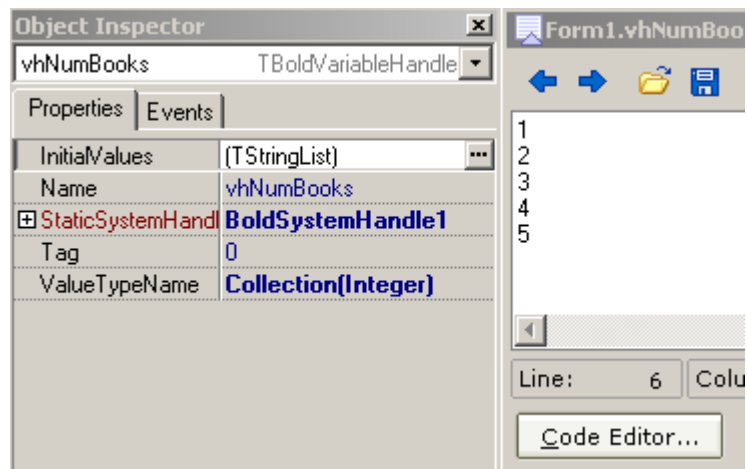


Fig. 7.13. Setting of initial values for OS-variable

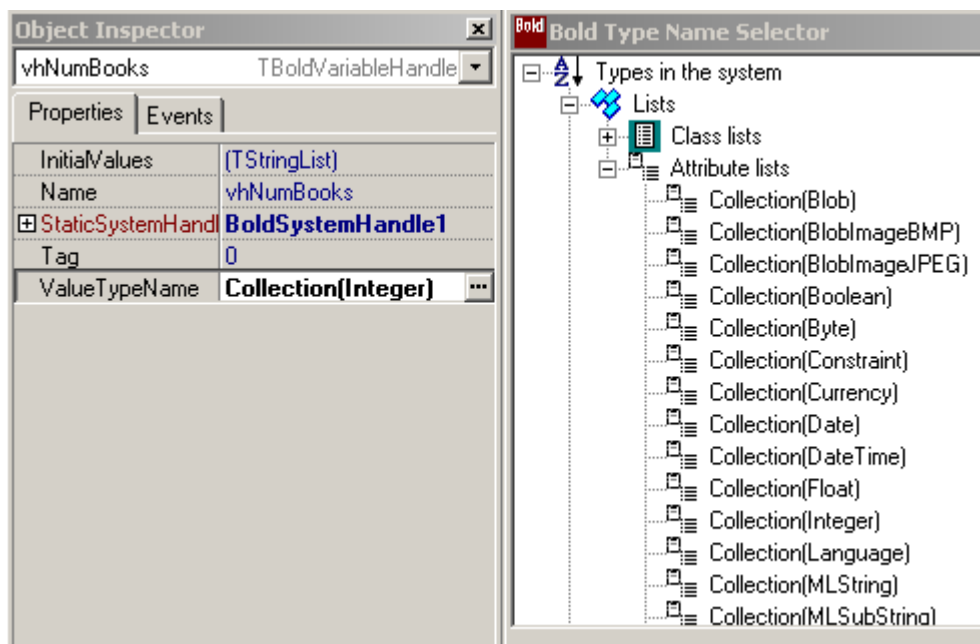


Fig. 7.14. Setting of the new type for OS-variable

And, at last, we shall add newly created cursor handle in container of OCL-variables - BoldOCLVariables (see Fig. 7.15), having given it an alias - curnum.

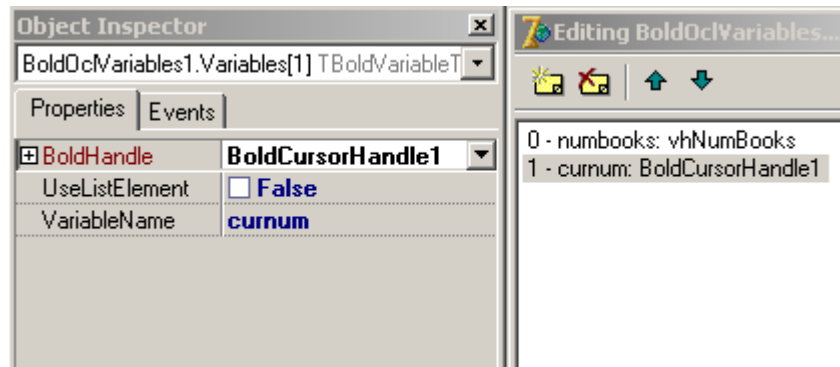


Fig. 7.15. Adding the cursor handle into the container of OCL-variables

Now it is necessary to change a little OCL-expression for BoldListHandle1 handle of authors list to the following:

```
if curnum=0
then Author.allInstances
else Author.allInstances->select(writes->size=curnum)
endif
```

And now we can start the new application for run in order to look, what for we did it. When selecting concrete figure (from 1 up to 5) on a new grid our list of authors is filtered automatically. At that the following change takes place: the information from an OS-variable, containing the set of initial values entered by us (1...5) due to connection of this variable to BoldCursorHandle1 new cursor handle “became available” for direct mapping on a grid. The point is that the grid demands only the list as a handle, and it cannot be connected to the OS-variable. The cursor handle just provides such transformation into the list. When selecting the string with concrete value on a grid this value is given automatically to our OS-variable, and OCL-expression processing with this new value takes place. In result we receive the filter, controlled from a grid.

As well as for BoldListHandle described in the previous section, BoldCursorHandle also always returns only a single value. This circumstance emphasizes a similarity of these handles types. And, certainly, the cursor handle also possesses all specific properties and methods for work with collections, as well as BoldListHandle (see Table 7.1 and 7.2).

BoldReferenceHandle

As follows from the name, this component-handle is intended for storage of the reference to any element of object space (BoldElement). This element should be set by StaticValueTypeName property by means of types selector already familiar to us (see, for example, Fig. 7.14). As the developers recommend, the given handle can be used as a starting handle of the application form. As well as in the previous case, the reference handle can be used in containers of OCL-variables, giving it an OCL-alias with the purpose of further use of this alias in OCL-expressions. Now we shall not show detailed examples of using this handle as the general principles and approaches of work with handles should be clear from the previous sections.

BoldUnloaderHandle

And, finally, we shall briefly consider the last from object space handles - a handle of objects unloading. Its purpose is to provide an automatic unloading of unused objects from OS after some time interval set by developer. Such functionality can be required in situations of lack of operative memory. Time interval set in MinAgeForUnload handle property (in milliseconds) serves as criterion of permission for objects unloading. This time interval sets the period, during which there was no reference to the object. The number of objects being simultaneously inspected for the purpose of unloading requirement is set by ScanPerTick property. It is not recommended to set large number for this property, because it will result in increasing pauses at carrying out of the periodical inspection. If this property is determined as 0, all OS objects will be inspected simultaneously.

The developer can make use of OnMayUnload and OnMayInvalidate events for writing procedures of their processing, in which it will be checked in addition whether it is necessary to unload concrete objects.

Chapter 8. Using OCL

Object Constraint Language Role in Borland MDA

In the previous chapter we have discussed handles of object space (OS). OCL is actively involved in practical use of OS handles.

In Borland MDA OCL plays extremely important role, carrying out the following basic functions:

- ❑ Navigation on model elements (classes, attributes, associations);
- ❑ Formation of OCL-queries to object space;
- ❑ Creation of derived attributes and associations.

Navigation on the model provided by means of OCL in Borland MDA, allows carrying out the flexible and powerful mechanism of queries to object space of the application. Such “OCL-queries”, as we shall see in this Chapter, basically are capable to replace completely SQL language usual for developers of databases applications (see also Chapter 5). Besides, taking into account platform independence of OCL, we can assert that these queries are universal, and are not bind to the concrete DBMS used in the application.

Conditional Model

For illustrating OCL use at work with OS handles we shall improve our application model having provided it with additional classes and associations (see Fig. 8.1). As a basis for new model we shall use the conditional model described in Chapter 5 at studying OCL language. We shall remind, that our application is intended for work with a library catalogue, and “Country”, “Publishing house” and “Subjects” classes are included in new model.

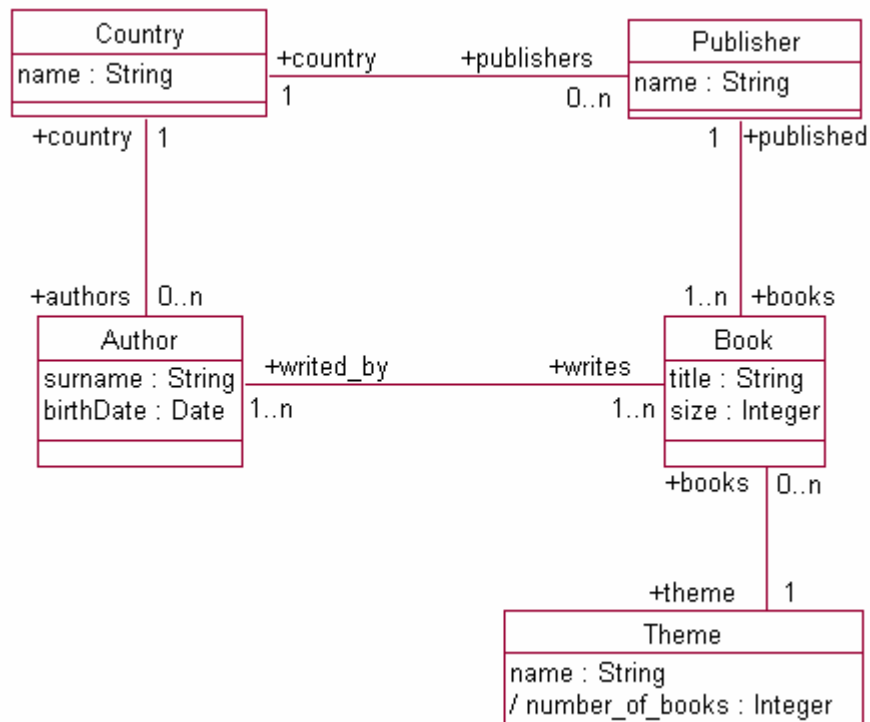


Fig. 8.1. The model being considered for studying OCL

Let's create the model in Rational Rose editor and save it in some folder on a disk with the name – lib.mdl, having processed it preliminary with the purpose of transformation of national language names of classes and attributes into English-speaking by transliteration of names.

Creating Application

Let's create the new project in Delphi consisting of one form and the data module. From components palette we shall place components on the data module and set their properties according to diagram given in Fig. 8.2.

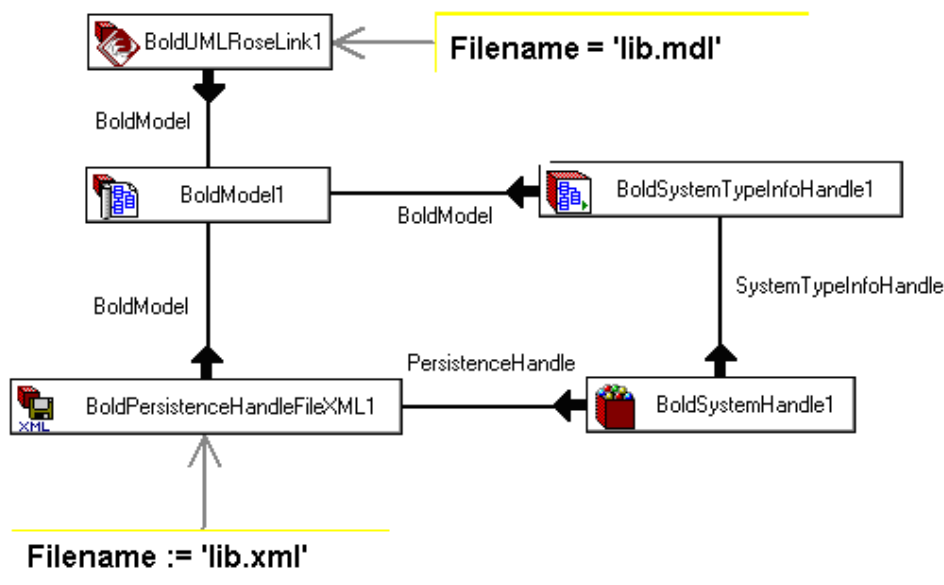


Fig. 8.2. Setting of properties of application components

Now we shall start creating GUI of the form (see Fig. 8.3). We shall place 7 visual components - BoldGrid1, BoldGrid2 ... BoldGrid7, on the form (in Fig. 8.3 their numbers are represented by large figures), then we shall place usual icons, one above each component for easy binding, having given to the icons the following names: “All authors”, “All books”, etc., as shown in Fig. 8.3. Under BoldGrid1,3,5,6,7 components we shall place BoldNavigator component, one for each. Further, we shall place three BoldLabel components (in Fig. 8.3 they are drawn out by a contour), without setting their properties for the moment (in Fig. 8.3 these components are shown in the adjusted view). We have created the GUI, however it is not bound yet to the object space. For realization of such binding we shall use not visual components – BoldListHandles from BoldHandles components palette, and we shall illustrate by the example how is OCL language used for interoperability of GUI (representation layer) and object space (business layer).

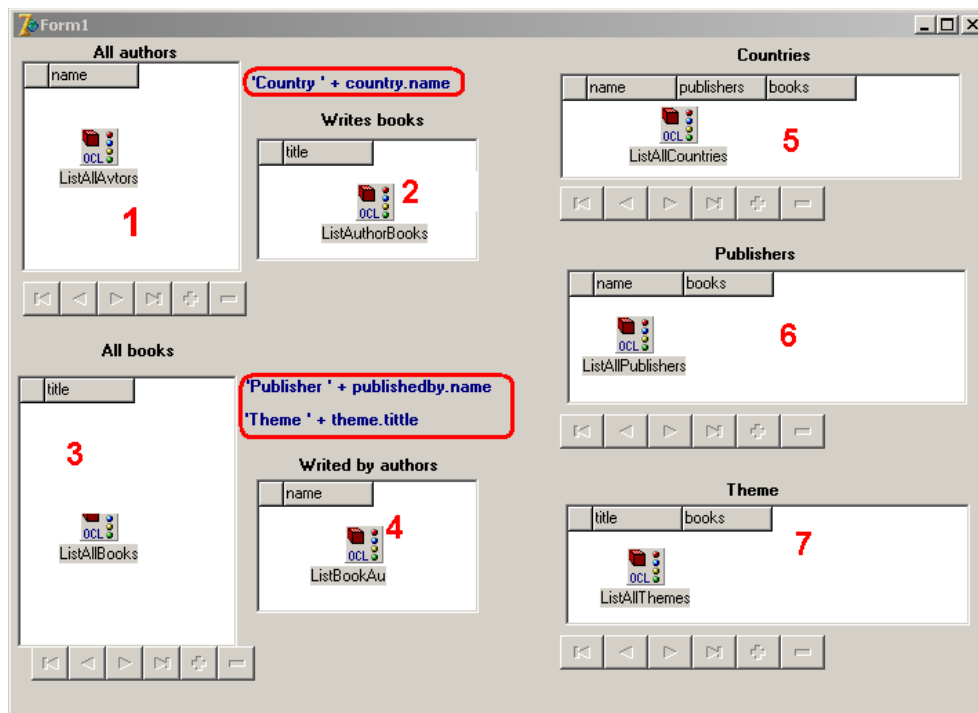


Fig. 8.3. View of demonstration application form

Built-in OCL-editor

Let's place BoldListHandle1 on the form, and give it ListAllAuthors name. The task of the given handle is obtaining the list of all available authors from object space, and transferring this list to BoldGrid1 visual component for mapping. The handle of the list concerns to the rooted handles (see the previous Chapter), i.e. it possesses <RootHandle> property, specifying a source of the information. In this case BoldSystemHandle1component – system handle of the whole object space represents such source. Therefore in the objects inspector we shall give DataModule1.BoldSystemHandle1 value to RootHandle property of ListAllAuthors component. Now we shall formulate OCL-query, which will provide the necessary information on authors. We will address to <Expression> property of our list handle. The given property describes expression in OCL, by means of which the necessary information selection is realized. This expression can be entered manually as we frequently did earlier considering an example of the simple application, however it is more convenient to take advantage of the OCL-editor built into Borland MDA. For its activation we shall double click on <Expression> property in the objects inspector, at that the window of the OCL-editor will be open (see Fig. 8.4).

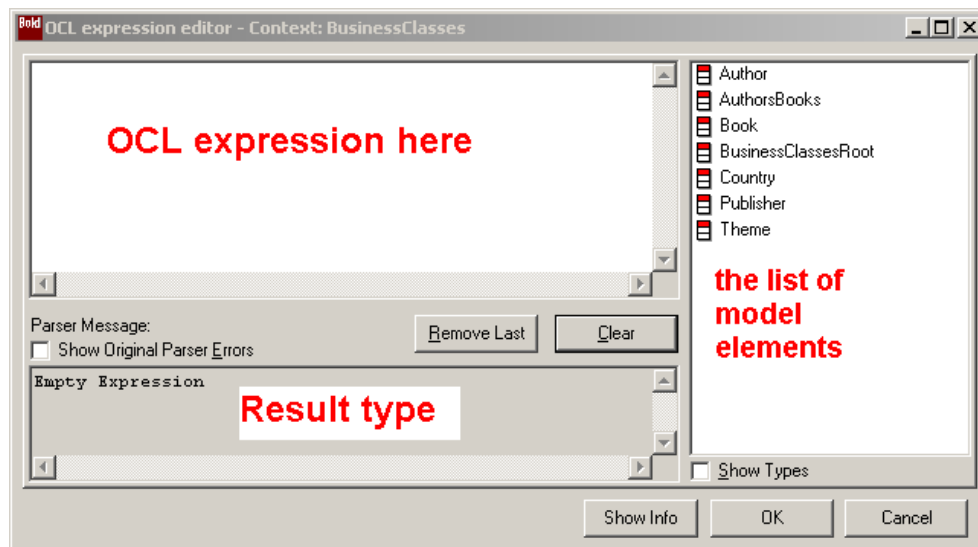


Fig. 8.4. Built-in OCL-editor general view

The OCL-editor interface is rather simple, in the top it has a window for entering expressions in OCL, under it the window for displaying the information on expression correctness and type of returned result is placed, and on the right there is a multipurpose navigating window. In this case until we formulate OCL-expression, the navigating window displays available elements of our model (in Delphi development environment they are represented as icons with a red strip). As we are interested now in authors, we shall double click in a navigating window on “Author” element and find out that the window of the OCL-editor has changed (see Fig. 8.5).

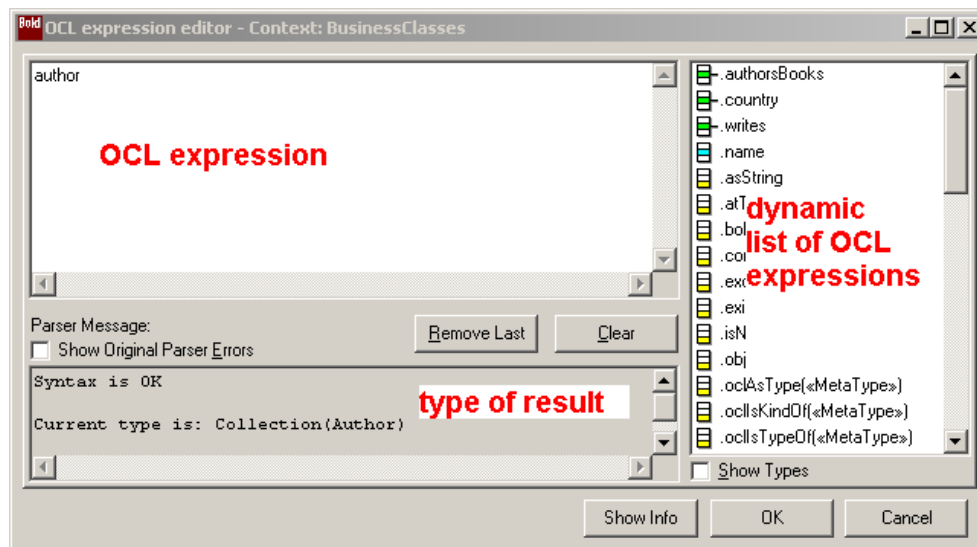


Fig. 8.5. Window of the OCL-editor at entering the class name

First of all, in the top window "Author" expression has appeared. Bottom window informs us that syntax of the OCL-expression is correct, and type of returned result represents metatype (that is model class). And at last, contents of a navigating window have considerably changed, now the list of the possible OCL-operators applied to our expression (icons with a yellow strip) is displayed in it. In the top of this list there is an operator we need – `<.allInstances>` (in this case all objects of `<Author>` type, as we have already set partly our OCL-expression). We shall double click on this element and find out a picture presented in Fig. 8.6.

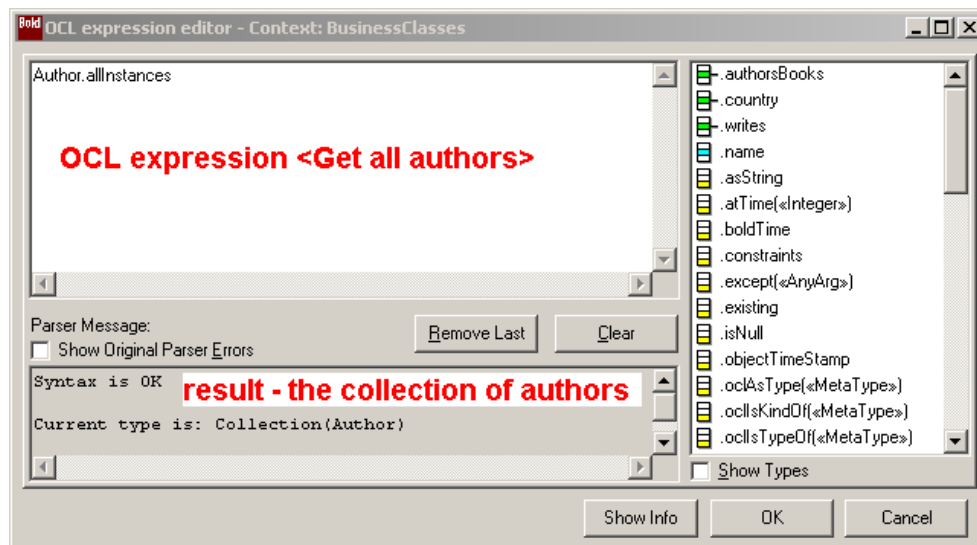


Fig. 8.6. OCL-editor window at entering OCL-expression

Returned result type now is represented by the collection of objects of <Author> type, which we need.

If we look attentively at the navigating window on the right, we can see a distinctive variety of elements mapped in it. Here are associations and their roles (icons with a green strip), not all of them, but only those connected with <Author> class. Also there are class attributes (icons with blue strip), in this case there is the single “Name” attribute of <Author> class. And, at last, as usual there are possible OCL-operators, which can “continue” our expression for working out in detail. Later we shall look how to use these wide opportunities, and now we shall note that in spite of seeming simplicity the OCL-editor is rather powerful tool for generating OCL-queries. Possessing “intelligence” and “knowledge” of model it dynamically forms possible variants of consecutive detailed elaboration of OCL-expression mapping them in the navigating window. Let's exit from the OCL-editor, and set BoldGrid1 and BoldNavigator1 components having set ListAllAuthors value to their BoldHandle property. Further, by means of the mouse right button we shall select “Create Default Columns” item on BoldGrid1 component from the pop-up menu; and we shall make sure that “Name” column heading has appeared. Thus, we have connected BoldGrid1 visual component to the object space by means of ListAllAuthors list handle.

Let's continue creation of our application. By analogy to setting of the list of all authors, we shall create ListAllBook list handle for obtaining the list of all books. The reader will do it easily without our help; the only difference is in OCL-expression, which in this case looks like “Book.allInstances”. The list of books we shall map in BoldGrid3, for this purpose we shall connect BoldGrid3 and BoldNavigator2 components with ListAllBook list handle in a way we have already known.

Forming Handles Chains

Till this moment as a matter of fact we have not made anything essentially new in comparison with the example of the simple application earlier considered in the previous parts, though we have begun to apply capabilities of the OCL-editor. In this section we shall get acquainted to technology of using chains of lists handles, and we shall show flexibility and power of OCL-queries. We shall pose the following problem: to map in BoldGrid2 books, not all books, but only those written by the concrete author, who is chosen in BoldGrid1. For solving this task we shall place a new handle of the list and give it ListAuthorBooks name. The basic moment in this case is the selection of a root handle – it is logically to select ListAllAuthors handle created by us instead of BoldSystemHandle1 system handle of object space used above. At that we form a chain of handles (see Fig. 8.7), in which each subsequent element refers on previous one by <Root Handle> property.

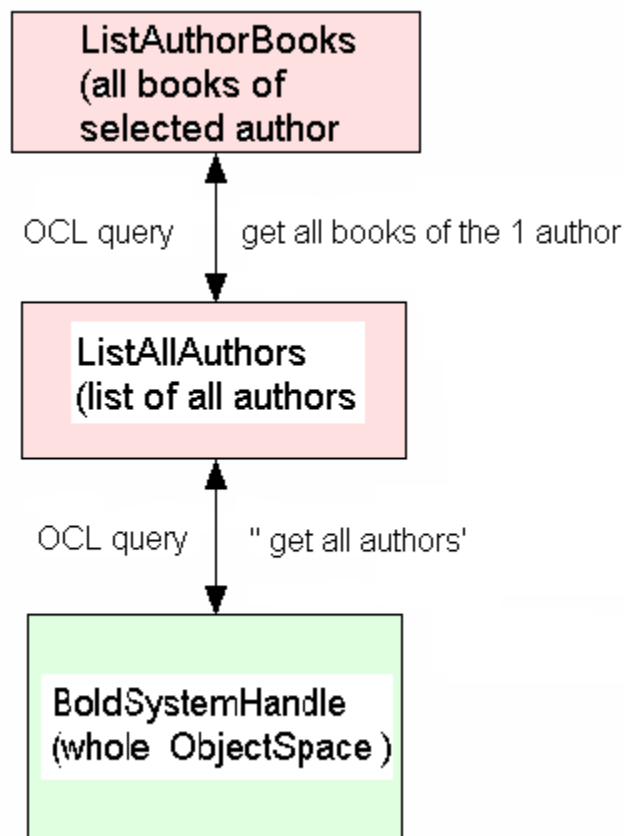


Fig. 8.7. Scheme of handles chain operation

Let's set ListAllAuthors value to <RootHandle> property of ListAuthorBooks handle in the objects inspector, and start the OCL-editor (see Fig. 8.8). Though we have not entered so far any OCL-expression, as it is easy to see, not all elements of our model are mapped in the navigating window, but only those concerned to <Author> class.

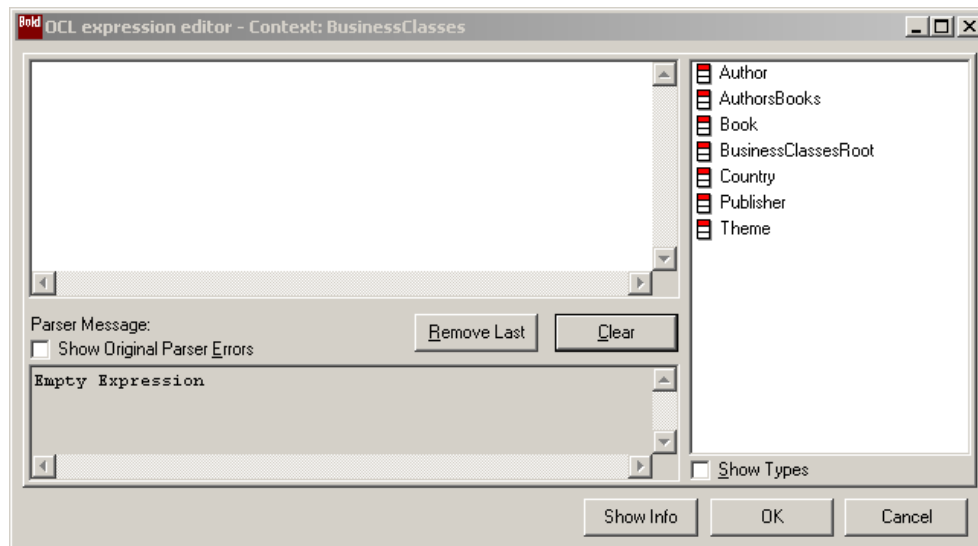


Fig. 8.8. View of OCL-editor window for the second handle of the chain

It is obvious; in fact we have selected not the object space, but only one its class as a root handle, i.e. the supplier of the information. Now it is necessary to double click on “has written” role name, and we shall make sure that the type of returned result is represented as objects collection of <Book> type. After BoldGrid2 connection to ListAuthorBooks handle it can be considered that the task is solved. Even by such simple example the flexibility given by Borland MDA environment is obvious. We shall continue work with our application. By analogy to lists of authors and books we shall connect BoldGrid5, BoldGrid6 and BoldGrid7 visual components to ListAllCountries, ListAllPublishers, ListAllThematics lists handles accordingly, that is we shall map in these grids lists of the countries, publishing houses and books themes (see Fig. 8.3). At that meanwhile in each of the specified grids we shall create columns by default (that is there will be only “Name” column in each grid). For data saving we shall write down in OnClose event handler of the form the following operator:

```
DataModule1.BoldSystemHandle1.UpdateDatabase;
```

Let's start our application, fill authors, books, countries, publishing houses and themes with the data, and we shall make sure that they are saved when exiting the application. Now we can get acquainted with the organization of business-layer and GUI interoperability in more detail.

OCL and GUI Link

Let's pose the following task: to map on BoldLabel1 the name of the country, in which the current author selected in a grid of authors (BoldGrid1) lives. BoldLabel1, as well as any other Borland MDA visual component, has BoldHandle property that is a Bold-handle. We have already known that it is expedient to select ListAllAuthors handle as a source of the data for this handle; so we shall do. In the form of OCL-expression we shall write manually or enter by means of the built-in OCL-editor (we shall remind, that the OCL-editor is called by double click on <Expression> property of ListAllAuthors component of BoldListHandle type) the following expression: <'Country' +country.name>, where 'Country' is a text constant, and <country.name> is an OCL-query for selecting the country for the current author and selecting 'name' attribute of 'Country' class - <name> for its mapping on our form. At that Borland MDA built-in mechanism will provide automatic modification of the label text when changing the current author in BoldGrid1. It is necessary to study more in detail this aspect; as it was mentioned earlier, the mechanism of so-called “events subscribing” is one of the most important Borland MDA built-in mechanisms. How does the given mechanism work?

In our case, having connected BoldHandle property of a label with ListAllAuthors handle, we “informed” Borland MDA environment that it is necessary “to subscribe” our label on the events taking place at any changes of the authors list (including the events occurring at transfers from one object to another when moving in the authors list in the grid). Besides we have generated OCL-query, which should be fulfilled at such event occurrence with the indication of the concrete information necessary for us (the name of the country, in which the author lives). Borland MDA environment will carry out the rest independently, that is at the event occurrence (change in the list of authors) it will check up, whether there are the elements, which have subscribed on change of this list, and if those are available, it will realize the corresponding OCL-query and will return the received data to the subscribed element for further use (in this case – for mapping on the form). Let's note, that in this case the environment carried out “the subscription”, and we didn't note this fact, but it is not of necessity. The developer can “subscribe” any Bold-element on necessary event on his own, having taken advantage of special methods of TBoldElement class.

The reader can ask: what is new here? In traditional development of databases applications there is also, for example, TDBLabel visual component – a label, which automatically keeps track and maps on the form contents of the specified field of the DB table when moving in the grid. However, new quality is represented here. Pay attention that in our “table” – an “Author” class – there is no “Country” “field”-attribute, and in order to solve this problem by means of traditional methods it would be necessary in general case to form SQL-query, which would select the country from other table for the concrete author, and a label to connect with this query. Besides it would be necessary to connect specially this query either to a grid or to a data set, so as it can be realized when changing authors. As it is easy to see, we have done without such operations.

Let's do operations on connection of BoldLabel2 (for mapping the name of publishing house of the book) and BoldLabel3 (for mapping subjects of the book) similarly. ListAllBooks books list handle acts as a source of the information for these labels. After the

application start we make sure that texts of labels keep track moving in the list of the books mapped in BoldGrid3.

OCL and Derived Data

Using OCL in Borland MDA allows providing flexible mechanisms for obtaining the various data. By the example of our application we shall consider some of these capabilities. For this purpose we will address to BoldGrid7 mapping subjects of books. Till this moment in the given grid there was only one user column mapping the name of subjects. Now we will put an additional task – to show total number of books of the given subjects in the same grid. For solving this task it is necessary to add one more column in a grid. For addition of a column we shall double click on a grid, at that the column editor will be mapped (see Fig. 8.9, at upper right). Addition of a column is carried out similarly, as well as at using Grid traditional component. We shall give “Total_of_books” name to the new column (it will be mapped in the column heading). However herein the analogy to a traditional component comes to the end. As we can see in Fig. 8.3, at the left, in the objects inspector well-known “Expression” property is present again. We shall enter into it “*books->size.asString*” expression manually or by means of the OCL-editor. In this case it means that it is necessary to take association role value of “Subjects” class with the “Book” name (see model in Fig. 8.1), and then to receive number of elements of this role – “*->size*” operator, and, at last, to transform the received expression into “*.asString*”. Thus we have solved the task.

Similarly, in BoldGrid6 we shall add a column for mapping the number of books issued by concrete publishing house. As it is easy to make sure in this case OCL-expression will be the same, since the role name is the same as in the previous case.

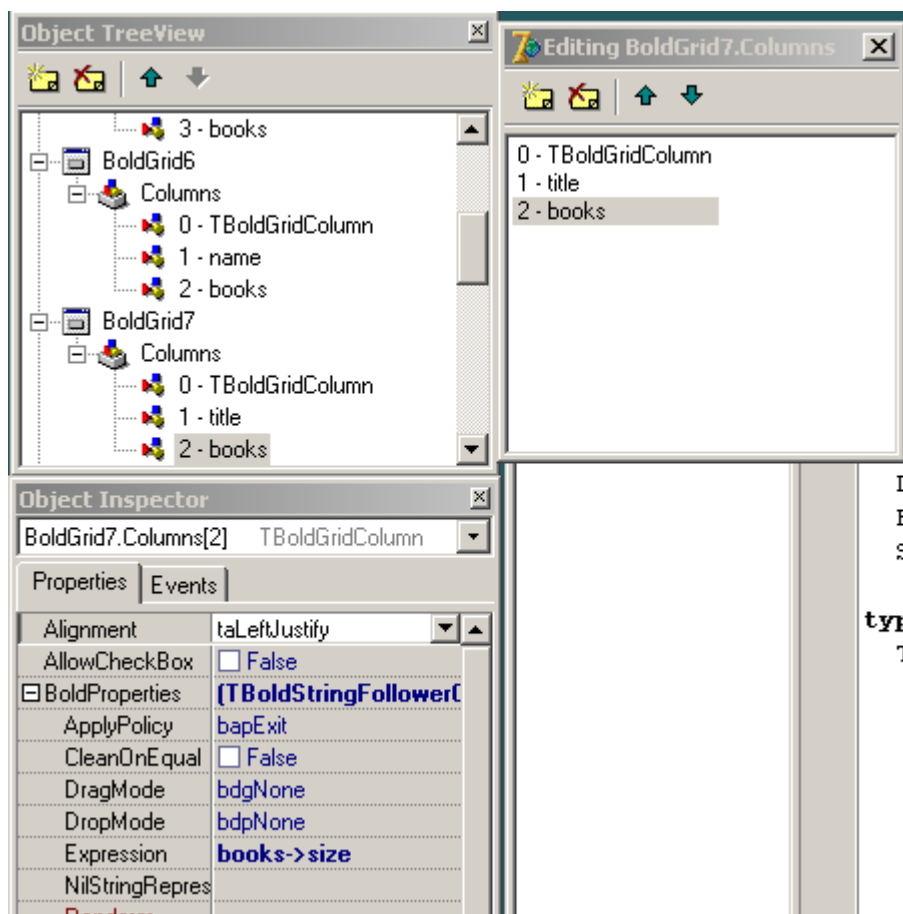


Fig. 8.9. Setting of BoldGrid component

And, at last, we will address to a grid mapping the names of the countries (BoldGrid5). Till this moment only one column created by default containing the name of the country was mapped in it. We shall assume that we want to receive the additional statistical information on each country, namely – total of the authors living in this country, total of publishing houses, and also total of all books issued in the given country. For this purpose we shall add to BoldGrid5 three new columns with the following names: “Authors”, “Publishers” (identical to Publishing Houses), “Books”. OCL-expressions will look like “*Authors->size.asString*” and “*Publishers->size.asString*” for “Authors” column and for “Publishers” column correspondingly. However for the last column such simple expression will not be the solution as the algorithm of calculating the total number of books is more complex. Really, in our model there is no the association directly connecting both classes: “Country” and “Book”, and it is not model shortcoming as we have got all necessary information for the task solution (see Chapter 3, where the similar example was examined).

For the given case the algorithm of the solution in a natural language can be formulated approximately so: for each country it is necessary to select all its publishing houses, to obtain the number of the books issued by each publishing house, and to sum all obtained values. In OCL the given algorithm can be realized by means of the following expression:

“Publishers->collect(books->size)->sum”

Here we meet again with “->collect” operators and “->sum” operator. Their meaning has been opened in Chapter 5 and is quite clear from the considered example. After the application start it is possible to make sure that it has got new qualities necessary for us.

Full source of the created application are resulted below (see Listings 8.1-8.3).

Listing 8.1. The Application Data Module

```
unit UDataModule;

interface

uses
  SysUtils, Classes, BoldPersistenceHandle, BoldPersistenceHandleFile,
  BoldPersistenceHandleFileXML, BoldHandle, BoldUMLModelLink,
  BoldUMLRose98Link, BoldAbstractModel, BoldModel, BoldHandles,
  BoldSubscription, BoldSystemHandle, BoldAbstractPersistenceHandleDB,
  BoldPersistenceHandleDB, BoldPersistenceHandleSystem;

type
  TDataModule1 = class(TDataModule)
    BoldSystemHandle1: TBoldSystemHandle;
    BoldSystemTypeInfoHandle1: TBoldSystemTypeInfoHandle;
    BoldModel1: TBoldModel;
    BoldUMLRoseLink1: TBoldUMLRoseLink;
    BoldPersistenceHandleFileXML1: TBoldPersistenceHandleFileXML;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DataModule1: TDataModule1;

implementation
{$R *.dfm}
end.
```

Listing 8.2. The Principle Module of the Application

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, BoldSubscription, BoldHandles, BoldRootedHandles,
  BoldAbstractListHandle, BoldCursorHandle, BoldListHandle, Grids, BoldGrid,
  StdCtrls, ExtCtrls, BoldNavigatorDefs, BoldNavigator,
  BoldLabel, BoldAFPDefault;

type
  TForm1 = class(TForm)
    BoldGrid1: TBoldGrid;
    BoldGrid2: TBoldGrid;
    BoldGrid3: TBoldGrid;
    BoldGrid4: TBoldGrid;
    ListAllAuthors: TBoldListHandle;
    ListAllBook: TBoldListHandle;
    ListAuthorBooks: TBoldListHandle;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    BoldNavigator1: TBoldNavigator;
    BoldNavigator2: TBoldNavigator;
    BoldLabel1: TBoldLabel;
    BoldLabel2: TBoldLabel;
    ListBookAuthors: TBoldListHandle;
    BoldGrid5: TBoldGrid;
    BoldGrid6: TBoldGrid;
    BoldGrid7: TBoldGrid;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    BoldLabel3: TBoldLabel;
    BoldNavigator3: TBoldNavigator;
    BoldNavigator4: TBoldNavigator;
    BoldNavigator5: TBoldNavigator;
    ListAllCountries: TBoldListHandle;
    ListAllPublishers: TBoldListHandle;
    ListAllSubjects: TBoldListHandle;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
```

```

        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

uses UDataModule;

{$R *.dfm}

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    DataModule1.BoldSystemHandle1.UpdateDatabase;
end;

end.

```

Listing 8.3. The Master Program Text

```

program Project1;

uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1},
    UDataModule in 'UDataModule.pas' {DataModule1: TDataModule};

{$R *.res}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TDataModule1, DataModule1);
    Application.Run;
end.

```

As it is easy to see, initial texts basically consist of variables declarations and external modules calls. Taking into account the received functionality of the application, it is impossible not to admit such exclusive briefness of program texts. When creating the application we practically did not write a code in Object Pascal language and did not use SQL, but, nevertheless, we could “tie” all elements of our model in a single whole. All basic functionality is not in the texts mentioned above, but consists in the calling external modules of Bold for Delphi environment. These modules provide obtaining the finished MDA-applications as a result of compilation. We shall remind here, that the created

application possesses all capabilities for editing the data, for access to which it is enough to double click on any element (the author name, the book title, etc.), after that windows of editing created by Bold environment will open automatically (autoforms, more in detail see Chapter 9).

By the example of the created application the basic approaches to realizing interoperability of business-layer and GUI have been shown. We made sure in practice that when using Borland MDA this environment provides essentially new capabilities for generating such interoperability due to the built-in OCL support. So, any visual element gets capability to map basically any data contained in objects of object space. In the considered example we have brought labels and grids to map the information from the objects belonging to different classes of our model, or even to receive the new information, which is not present in model directly (the statistical data on the countries, publishing houses, etc.). It is necessary to note that formation of such interoperability is not limited to the application development stage. Any OCL-expression can be formed and assigned during the program run too, and such OCL-query will be realized by Borland MDA environment as the OCL-interpreter is included in structure of an executed exe-file of the application the same way as all information on model.

Using OCL allows making the application more “platform-independent” as OCL itself is platform-independent language. In practice it means that the application business-logic, being created once in OCL without coding in programming languages, when carrying development on another platform is kept completely and it can be used repeatedly. And this is one more advantage of Borland MDA technology.

Using OCL in Program

Formation of OCL-expressions is possible not only in properties of Bold for Delphi components. There are situations, when it is necessary, and can be sufficiently simply realized directly in the application code. We shall consider a simple example.

Let it is necessary for us to set concrete type of the information in a grid on the form depending on pressing various radiobuttons (see Fig. 8.12).



Fig. 8.12. The simple application

I.e. by pressing the radiobutton with “Authors” heading the list of authors should be mapped in a grid; by pressing the radiobutton with “Books” heading the list of books should be mapped in a grid, etc. How can we do it in the most simply way? Certainly, we can place three handles of the list on the form, and connect a grid to any of them depending on the radiobutton chosen by the user. But now we shall act in another way to show interoperability of a code and OCL, and at the same time to get acquainted with some nuances of UML-model elements setting in BMDA.

We shall take as a basis the application considered above (datamodule and model) and create the new simple form (see Fig. 8.13), containing BoldGrid, BoldNavigator, List handle, and three radiobuttons, top from which (Authors) we shall set as checked. As OCL-expression for List handle we shall enter

`Author.allInstances`

in order to map by default the list of authors.

Let's write the following event handler for radiobuttons (see Listing 8.4).

Listing 8.4. Forming OCL-expressions in Program Code

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
    List.Expression:='Author.allInstances';
end;
```

```

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
    List.Expression:='Book.allInstances';
end;

procedure TForm1.RadioButton3Click(Sender: TObject);
begin
    List.Expression:='Country.allInstances';
end;

```

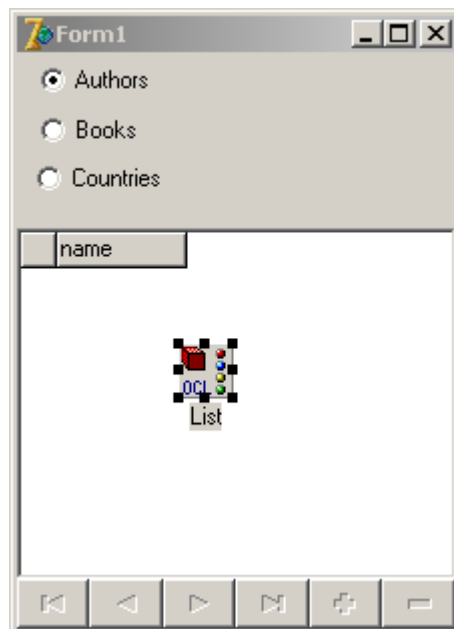


Fig. 8.13. The application form

From the given code it is clear that Expression property is the usual text, which can be formed in program way. If now we start our application it will obediently change classes by pressing various radiobuttons, however at that mapping the information in a grid will be a little bit strange (see Fig. 8.14).

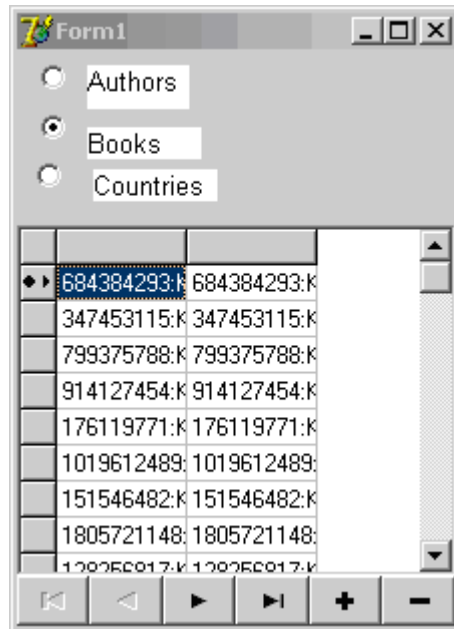


Fig. 8.14. View of the grid without setting of the class representation

It can be explained simply. In this case we did not adjust a grid for mapping “by default”, as in the previous examples. Therefore it “does not know” what exactly is required to be mapped, and shows the internal information of system on objects identifiers. In order that any visual element “knows” without its additional adjustments what exactly to map for the concrete class, there is a special parameter for each model class named “Default String Representation” - i.e. text representation by default. For adjusting this parameter we shall open the model built-in editor, select “Author” class in the model tree, and enter “Name” OCL-expression (an attribute – surname of the author) into the window on the right, denoted Default String Rep (see Fig. 8.15). Similarly for “Book” and “Country” classes we shall enter “Title” and “Name” expressions correspondingly. It is possible not to enter these expressions manually, and use the built-in OCL-editor, which in this case is called by pressing the button located nearby with a question-mark. For the classes containing variety of attributes, realization of the mentioned mechanism allows to set the most typical attribute by default, which will be mapped by all BMDA visual components.

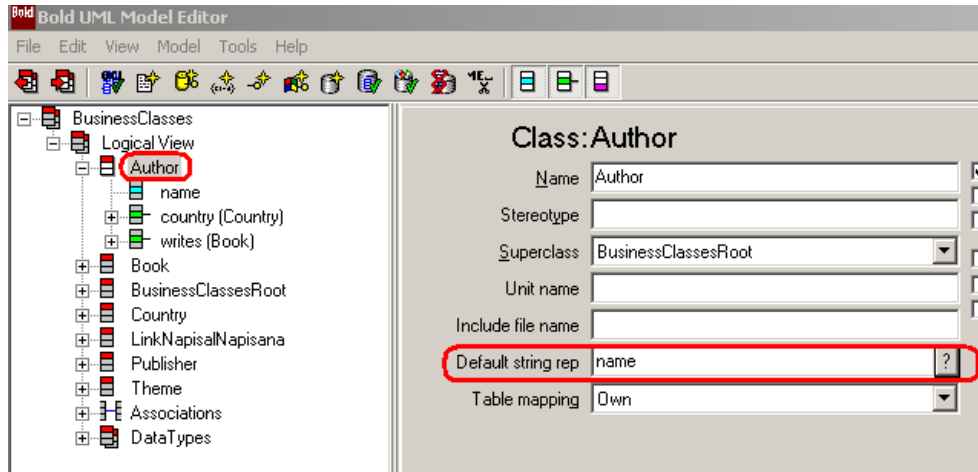


Fig. 8.15. Setting of representation parameter by default

After the described settings our application will function nominally (see Fig. 8.11). Thus, by very simple example we make sure that if necessary it is possible to form OCL-expressions directly in the application code.

Program Work with TBoldElement Class and Its Successors Using OCL-derivations

In Chapter 6 TBoldElement class being a parental class for all model elements was considered. The methods, which effectively use OCL, are among its numerous methods. Here we shall show by simple examples how to use in practice some capabilities of this class. Coming back to the application created earlier (see Listings 8.1-8.3) we shall put the following task – to calculate total of authors in the catalogue in program way without using handles. We have already known that for obtaining such information it is necessary to derivate OCL-expression of the following type:

Author->allInstances->size

At that OCL-expressions interpreter built-in into an executed file of the application will provide necessary derivations without using program coding.

We will show how to take advantage of capabilities given by OCL-interpreter in program way. For this purpose we shall use EvaluateExpressionAsString method of TboldObjectList class, which is the successor of TBoldElement class (see Chapter 6). Let's write the following OCL-expression:

```
i:=strtoint(datamodule1.BoldSystemHandle1.System
```

```
.ClassByExpressionName['Author'].EvaluateExpressionAsString('self->size',1));
```

where *i* – is the integer variable, into which the required total of authors is transferred. In spite of inconvenient view of the represented expression, its meaning is rather simple, and it was partly considered in Chapter 6. “Novelty” in the given expression is a call of `EvaluateExpressionAsString` method with text parameter – OCL-expression of “self->size” type. In this case `self` means the current context (see Chapter 5), determined by ... `ClassByExpressionName['Author']` expression, i.e. a collection of objects of “Author” type. Feature of similar application of `EvaluateExpressionAsString` method is that it is not bound to any handles, or visual Bold-components, but depends only on knowledge of model by Bold environment and the developer of a program code. It is natural, that much more complex OCL operators can be used as OCL-expressions, thus obtaining a capability of complex expressions derivation without programming in Object Pascal language. Such capabilities can be successfully applied, for example, if it is necessary to provide mass derivations of the summary analytical data for output of the reporting information. Other possible application is derivation of “pseudo-calculated” attributes with their subsequent saving in a database (we shall remind that derivable attributes in a database cannot be saved by usual way).

OCL-repository

OCL formation in the program text is similar to the process described in the previous section; it provides the developer with flexible capabilities, for example, at dynamic formation of OCL-expressions, or even for their generation in passing. However such practice should not be abused, as in result integrity of the application is broken, and the part of business-logic contained in OCL, “gets smeared” over a code. It can result in bad readability of the program and in difficulties when changing language platform. Probably, having in view such constraints, BMDA developers have included a so-called OCL-repository, which, as a matter of fact, is storehouse for various OCL-expressions, in their system composition. We shall show the operations with this component by the example from the previous section. We shall place `OCLRepository1` component from `BoldMisc` bookmark on the form, and give it “OCL” name (see Fig. 8.16).

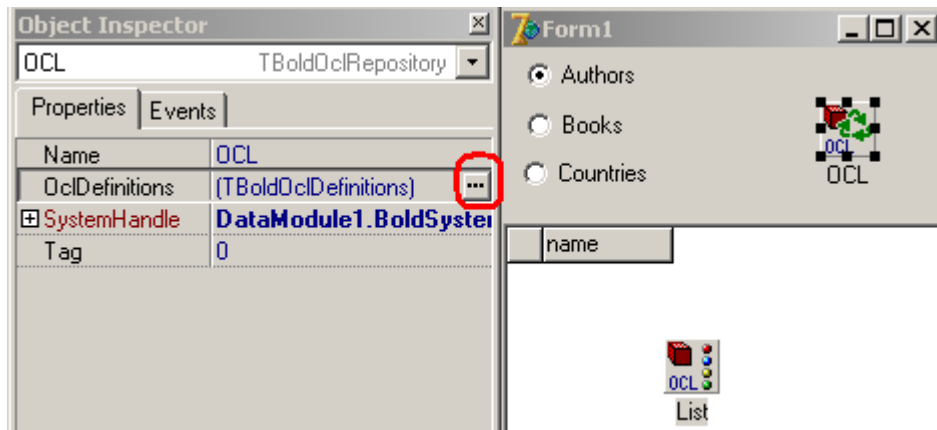


Fig. 8.16. Form with OCL-repository and component properties

In the objects inspector we shall set for it DataModule1.BoldSystemHandle1 as “SystemHandle” property, then we shall open OclDefinitions properties by pressing the button with dots (see Fig. 8.16, 8.17).

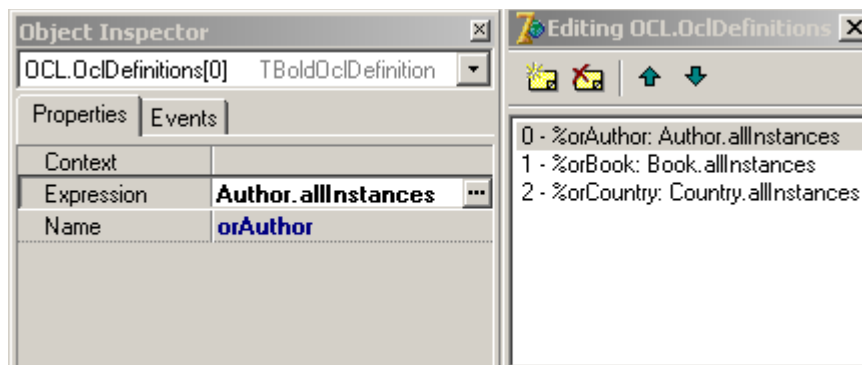


Fig. 8.17. Editing OCL-elements composition

In the opened window of editing repository structure we shall add three elements, and for each element in the objects inspector we shall give aliases (Name property): “orAuthor”, “orBook”, “orCountry”, and then we shall enter the same OCL-expressions, which in the previous example we gave in a code (see Fig. 8.16). We shall rewrite radiobuttons pressing handlers according to Listing 8.5.

Listing 8.5. Setting of OCL-expressions from repository

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
    List.Expression:=OCL.LookUpOclDefinition('orAuthor');
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
    List.Expression:=OCL.LookUpOclDefinition('orBook');
end;

procedure TForm1.RadioButton3Click(Sender: TObject);
begin
    List.Expression:=OCL.LookUpOclDefinition('orCountry');
end;
```

In the above-stated example of a code to the concrete OCL-expression contained in repository, the access is realized by its alias using LookUpOclDefinition property. Also for this purpose it is possible to apply OclDefinitions.Items [index] property (where index is a number of expression) to access the necessary OCL-expression by number.

Summary

In this Chapter work with OCL in Bold for Delphi environment is described. Flexibility and power of joint using OCL and object space handles and also GUI are shown. On the other hand, it is shown on the concrete examples how to use OCL capabilities directly from a program code, achieving at that new quality, namely, involvement of functional computing capabilities of the built-in OCL-interpreter without programming complex expressions at the code layer.

Chapter 9. Graphic User Interface – GUI

Bold for Delphi development environment includes proper visual components, intended for generating the graphic user interface (GUI), and also special control means of the visual information display – *renderers*. The general structure of the application business-layer and GUI interoperability can be described as follows. The information from business-layer, which availability is provided by means of object space handles (see Chapter 7), is transferred to the representation layer (the graphic interface) by means of renderers (see Fig. 9.1). Renderers operate the process of the information display, forming concrete visual data presentation on the basis of rules set by the developer.

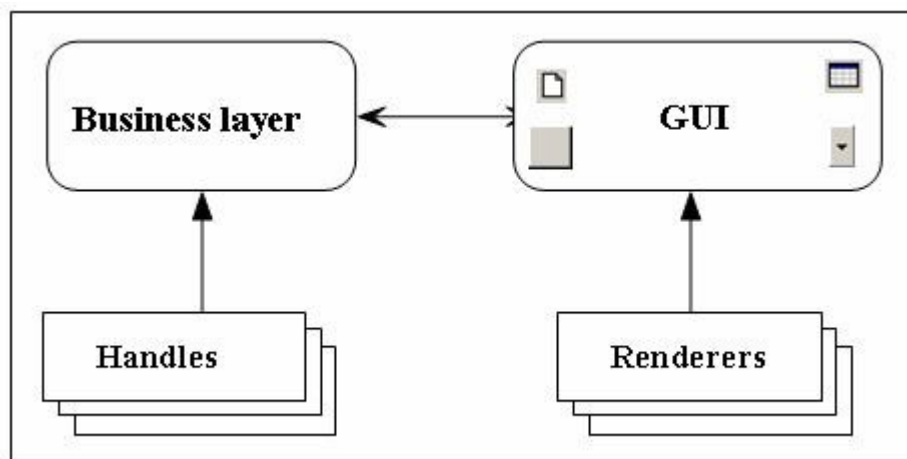


Fig. 9.1. The general scheme of the business-layer and the graphic interface interoperability

In the given chapter the description of Bold for Delphi visual components is given and problems of visual information display control by means of renderers are considered

Visual MDA-components Features

In Borland MDA for Delphi 7 version under consideration, the visual components intended for use in MDA-applications, have a number of specific features. It results from necessity of connection of these components to the application object space (OS), which as we have already known, has proper control mechanisms of objects behaviour and processing the various events taking place in system.

BoldHandle property being subproperty of BoldProperties property and representing the OS-handle (a handle of object space) is the most common property available at each visual MDA-component. Purpose of such handle is providing the information for display in a visual component. For indicating the concrete information content obtained from the OS-handle each visual component has also Expression text property meant for setting of OCL-expression. Further in this section we shall consider the basic graphic MDA-components and features of their practical use.

BoldLabel

This component is the simplest one from the graphic MDA-components. Its task is to display the information in Caption property, turning out as a result of calculation (estimation) of the set OCL-expression. Use of the given component has been shown in the previous examples and does not cause difficulties. Here we shall consider only one interesting feature – BoldLabel, namely a fundamental opportunity of “Drag-n-Drop” interface involvement. For this purpose on the basis of the examples made earlier (see Chapter 3) we shall create the simple application consisting of data module created earlier and one form, on which we shall keep BoldGrid to display the list of authors, and in addition we shall place BoldVariableHandle1 components and BoldVariableHandle1 OS-variable handle (see Fig. 9.2).



Fig. 9.2. The application form

For OS-variable handle we shall set a variable type – “Collection (Author)”, as it shown in Fig. 9.3.

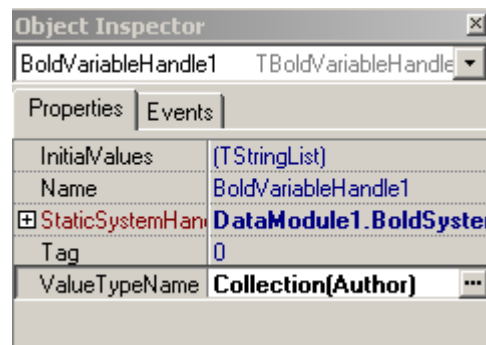


Fig. 9.3. Setting of the variable handle

For BoldLabel we shall set the following properties (see Fig. 9.4):

- ❑ BoldHandle = BoldVariableHandle1;
- ❑ Expression='self.name->first';
- ❑ DropMode=bdpInsert;
- ❑ NilStringrepresentation='No author'

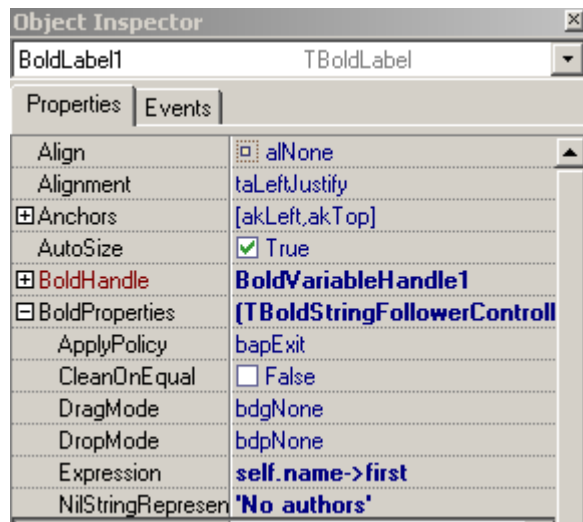


Fig. 9.4. Setting of BoldLabel properties

Let's start the application run. We shall make sure that our label displays “No author” text by default. Now we shall drag with the help of the mouse any author on the label. We shall see that the label text began to display a surname of the dragged author. Thus, due to this example we understand that even the simplest MDA-components are capable to interoperate actively with other MDA-components and at that to support “Drag-n-Drop” interface practically without participation of the programmer.

BoldEdit

This component is also the elementary MDA-analogue of usual TEdit OCL-component. Its purpose is input and editing of the text. BoldEdit possesses <ApplyPolicy> new property, which many MDA-components have got. This property can posses three various values. BapExit value is set by default; it means that all changes made in the editing window will be recognized by environment only after loss of focus by a component. If this property is set in bapChange, then any change is taken by environment at once. And, at last, if property ApplyPolicy is set in bapDemand value change will not be recognized until special BMDA methods – T-BoldQueueable.ApplyAll or T-BoldQueueable.ApplyAllMatching for OS updating will be called explicitly. Last capability is useful, when necessity occurs for the caching of done changes, and their fixing in OS on the event appointed by the developer, for example, by pressing the button – “To keep changes” on the form of the application.

BoldGrid

We used this component in examples more than once. It is MDA-analogue of DBGrid component. However BoldGrid capabilities go beyond the last one. As we have already made sure (see Chapter 8), the given component allows to set individually the information displayed in every concrete column, at that effectively using OCL-navigation on UML-models. Therefore, “having bound” BoldGrid to one model class we receive an opportunity to display the information in its various columns from other classes connected with a “source” class through associations-links. These capabilities allow “to collect” all necessary information in columns of the single BoldGrid component not applying SQL-queries. We shall illustrate the capabilities described above without repeating the cases considered in detail in Chapter 8 with the following example. Having bound BoldGrid by means of the list handle to “Author” class, which is connected in our UML-model by association to “Country” class, we can easily display the name of the country for each author in the second column of BoldGrid (see Fig. 9.6). For this purpose it is enough to set navigating OCL-expression of “country.title” kind in the second column “Expression” property. And if we want to display number of the books written by each author in one more column of the same grid, we can enter for this column “has written->size” OCL-expression. And so on. This capability of BoldGrid component is very convenient, as the developer gets an opportunity to receive practically any available information in one BoldGrid component without programming a code and SQL-queries. Further we shall show some features not considered earlier.

BoldShowConstraints Property

This logic property gives the developer a convenient opportunity to control visually model constraints violations. If this property is set in True value, then an additional column occurs in BoldGrid , in which special multi-colored indicators are displayed. If for the given element (string) the constraints set in the model are violated, color of the indicator is red

BoldHandleIndexLock Property

This logic property controls interoperability between visual component and the list handle, to which the given component is connected. If we set True value for this property, the cursor updating on strings of a grid will automatically call the index moving in the list handle. Sometimes it is rather conveniently for synchronization of all other visual MDA-components connected to the same handle or to the other handles, for which the given handle is root (see Chapter 7 concerning handles chains). At that it is not important, whether the synchronizable visual components are located on the same form, or on the other forms of the application.

DefaultDbClick Property

This logic property controls BMDA autoforms display. If we set False value for this property, autoforms will not appear. We shall remind that for autoforms display it is also necessary to add the module named “BoldAFPDefault” into the list of used modules (USES).

BoldSortingGrid

In structure of visual components there is “an advanced” equivalent of BoldGrid component. It is known as BoldSortingGrid and is located on <BoldMisc> bookmark of components palette. Distinctive feature of this component is the built-in capability of records ascending and descending sort, which is carried out by mouse click on the column heading . At that sorting is realized according to the records set displayed in the given column.

In all other respects using BoldSortingGrid component practically does not differ from “usual” BoldGrid component considered above.

BoldComboBox

The given component is intended for selecting one element from some pull-down list and assigning its value to the element of the other list. From this point of view it is possible to say that BoldComboBox is something like analogue of known DBLookupComboBox component. The basic properties of the component are two references to OS-handles. BoldHandle property specifies the list handle, which element will change when selecting the concrete value from the pull-down list. BoldListHandle property specifies the list handle, which elements will be displayed in the pull-down list. We shall consider a simple

example for an illustration. Using the application created earlier, we shall keep BoldGrid on the form to display the list of authors, and add BoldComboBox1 component (see Fig. 9.9).



Fig. 9.9. View of the form with BoldComboBox component

We shall set component properties as follows (see Fig. 9.10):

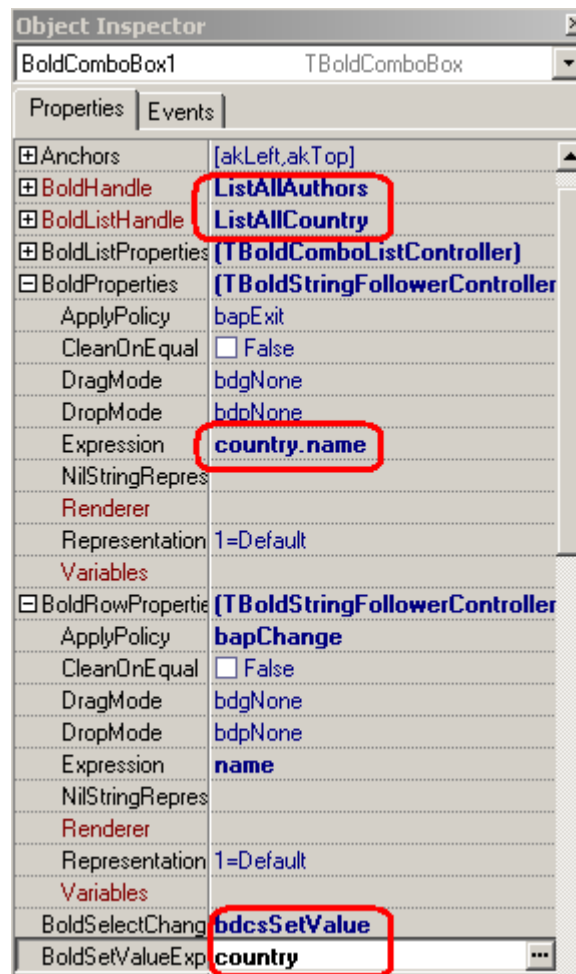


Fig. 9.10. Setting of BoldComboBox component properties

- ❑ BoldHandle=ListAllAuthors (the list of all authors);
- ❑ BoldListHandle=ListAllCountry (the list of countries; the names of countries will be displayed in the pull-down list of the component);
- ❑ BoldProperties->Expression='Country.name' –BoldComboBox when moving in the list of authors in BoldGrid;
- ❑ BoldRowProperties->Expression='name' - this OCL-expression determines, what information will be displayed in the pull-down list of the component;
- ❑ BoldSelectChangeAction = bdcsSetValue – this property sets the type of the value transmitted from the pull-down list; in this case this is the value itself (it is also possible to transmit index in the list, reference or text);

- ❑ `BoldSetValueExpression = 'country'` – this OCL-expression determines, what will be transmitted as an element for assignment; in this case this is an object of “Country” type.

Let's start the application. First, we will see that at moving in a grid and selecting the concrete author, in `BoldComboBox` component the name of the country changes according to the given author. And, second, having selected the concrete author in a grid, we can open the pull-down list, then select any of the countries, and thus “to appropriate” this country to the current author (see Fig. 9.11).

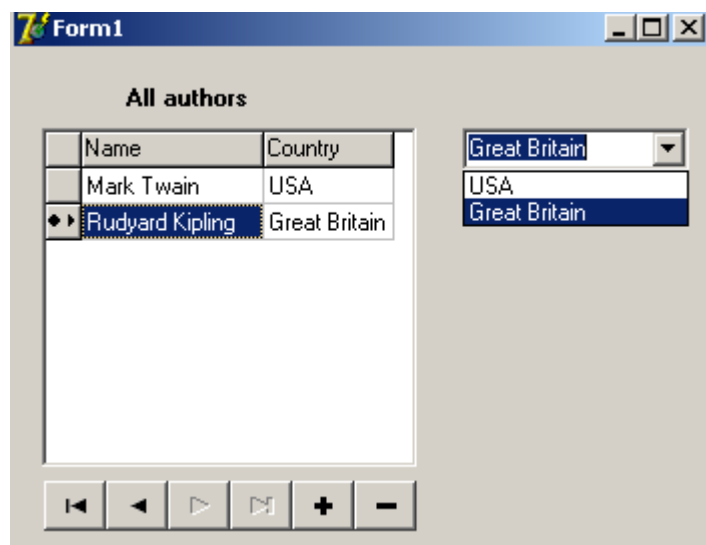


Fig. 9.11. Setting of the country of the author residence by means of `BoldComboBox`

BoldListBox

The given component is MDA-analogue of standard `TListBox` component. Its use is trivial enough. But it is necessary to remember that `BoldListBox` has `BoldRowProperties` property, as well as the previous component. In this property the OCL-expression determining what exactly is displayed in the list is set. As well as usual `Listbox` Delphi-component, `BoldListBox` also is capable to display the data, locating them in several columns. And, naturally, after double click on any record in the list, the autoform will open for data viewing and editing, concerning to this record, in the same way as in the case of `BoldGrid` component.

NOTE

For opening autoform it is necessary to set “`DefaultDBLClick`” property in “`True`” value. Besides `BoldAFPDefault` module should be added in structure of connected modules (“`Uses`” section).

BoldCheckBox

This component is intended for display and editing of logic values such as Boolean. When using BoldCheckBox, it is necessary to distinguish situations, when the capability appears to edit value displayed by it, and cases, when this component is capable only to display logic result. We can be guided by the following simple rule: visual editing of logic value is possible only in the case, when the given component directly displays a condition of logic attribute of any class. Otherwise BoldCheckBox component will display only “validity” of the current OCL-expression set by “Expression” property of the given component. For example, if in the above-stated example of the simple application we place BoldCheckBox component on the form, connect it with the authors list handle, and set “name<>'Mark Twain” OCL-expression (that is the author is not Mark Twain) as its “Expression” property, then we shall obtain a component “only for reading”, which will display checkmark only when selecting author Ilf in the list of authors.

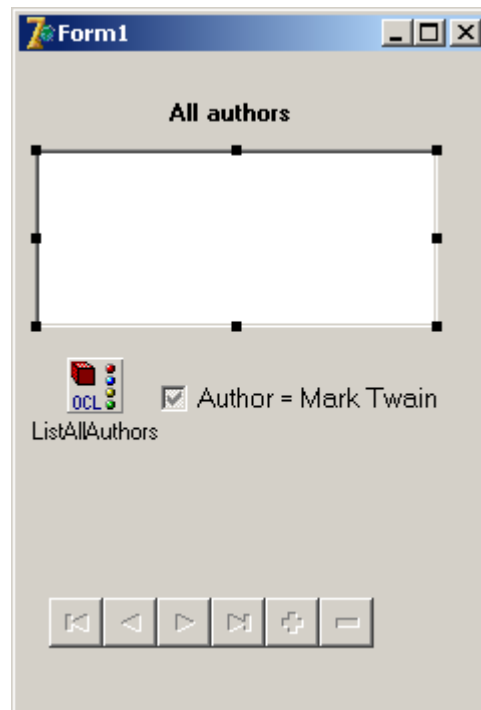


Fig. 9.13. BoldCheckBox “for reading only”

The attempt of changing the component state (uncheck) will not result in success in such situation. It is clear, as in this case such change would mean compulsory change of other attribute - surname of the author, to whom BoldCheckBox “has not any concern”. And on the contrary, if we add new logic attribute in structure of attributes of “Author” class, for example, a sex (male=True, female=False) and indicate it in OCL-expression of “Expression” property of considered component, then in this case we can directly edit a condition of such attribute.

BoldTreeView

BoldTreeView component is the most complex of visual Bold-components. As it is easy to guess according to its name, its purpose is to display a set of some elements in the form of “tree”, i.e. hierarchical structure. It is simple to understand principles of work of this component by the concrete example. For this purpose we shall use UML-model created earlier (see Chapter 8), which fragment is shown in Fig. 9.17.

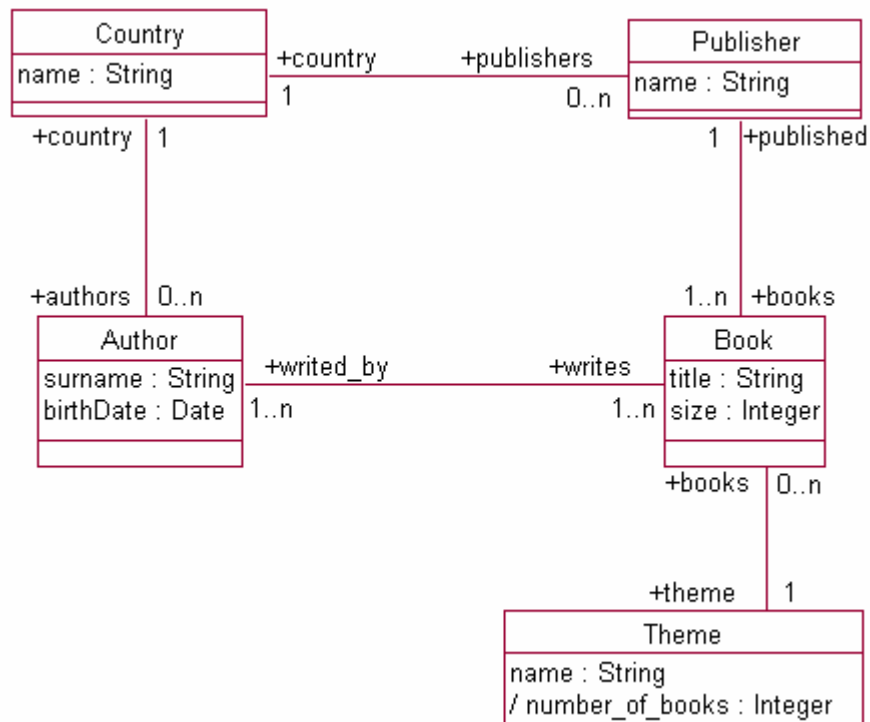


Fig. 9.17. UML-model fragment

In our example we shall use “Country”, “Author” and ‘Book” classes from the set of model elements. We shall set the following task: to display in the form of “tree” the authors living in the chosen country, and the books written by each author, at that books should be connected to each author in the hierarchy tree (see Fig. 9.18).

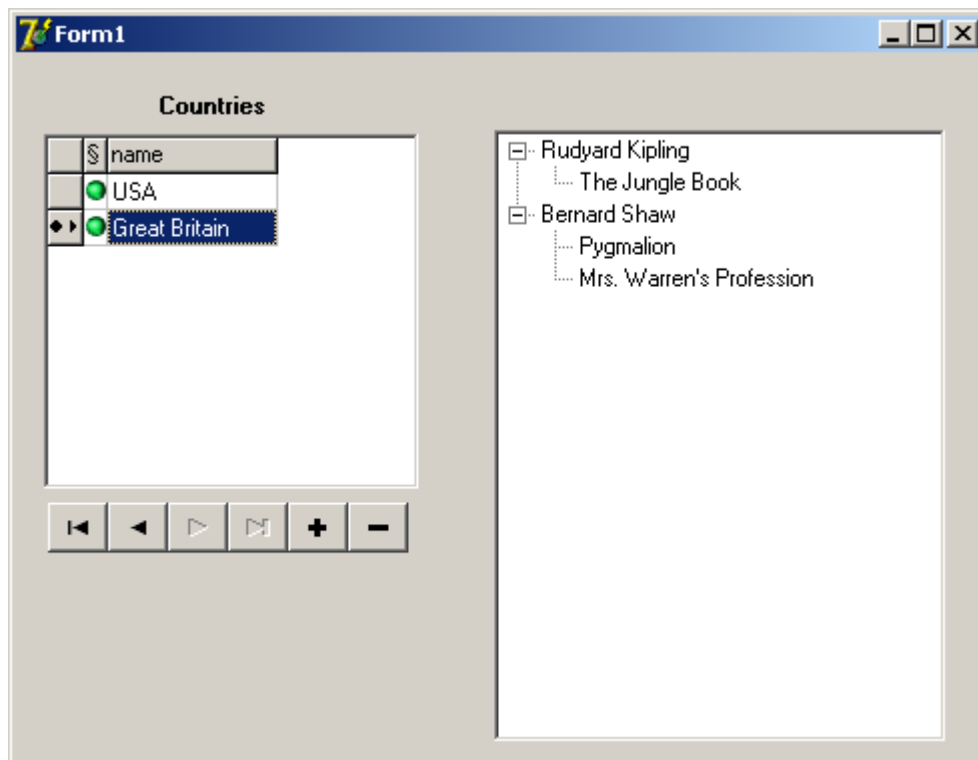


Fig. 9.18. Application final form

Let's place BoldListHandle component on the form, name it "ListCountry" and link it to the set of the countries, having entered "country.allInstances" OCL-expression as its "Expression" property. We shall place DBGrid component on the form, and connect it to ListCountry handle. And, at last, we shall add BoldTreeView1 component on the form, and connect it (BoldHandle property) to ListCountry handle too (see Fig. 9.19).

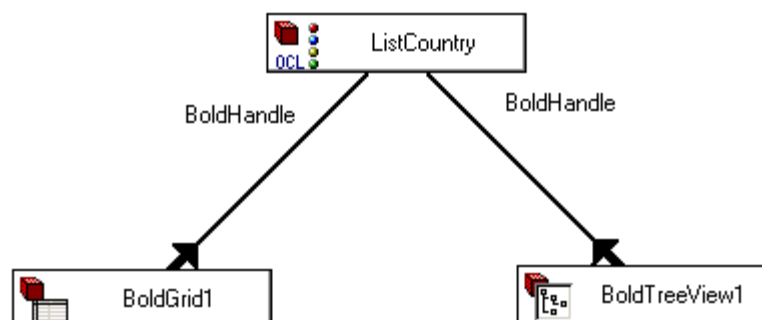


Fig. 9.19. Diagram of application components adjustment

Thus, BoldTreeView component will have a collection of the countries as a source of the information. We shall solve the task stage by stage. At the first stage we shall try to get the situation, when only authors of each country without books are displayed in the tree. For setting of our “tree” properties we shall double click on BoldTreeView component, at that we will see the window of the properties editor for this component (see Fig. 9.20).

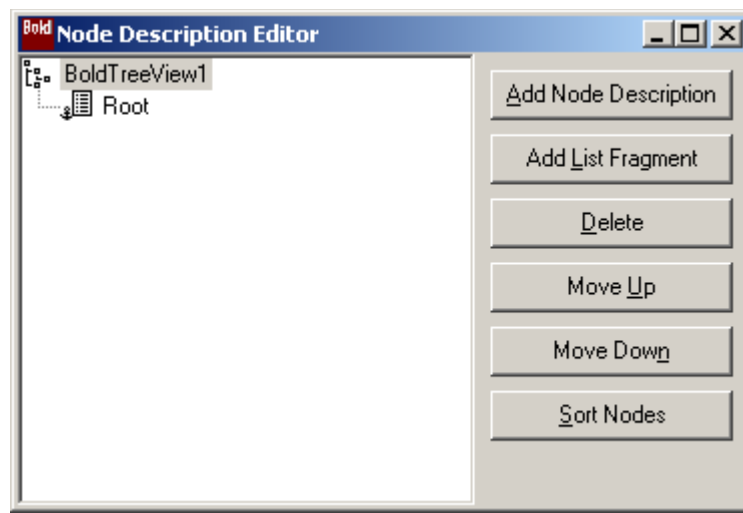


Fig. 9.20. The properties editor of BoldTreeView component

Let's add an element-description of a new layer of hierarchy, which will be responsible for authors of books display. For this purpose we shall press “Add List Fragment” button, located on the right in the properties editor, and we shall see that under “root” of our “tree” the new element has appeared. We shall adjust its properties as follows (see Fig. 9.21).

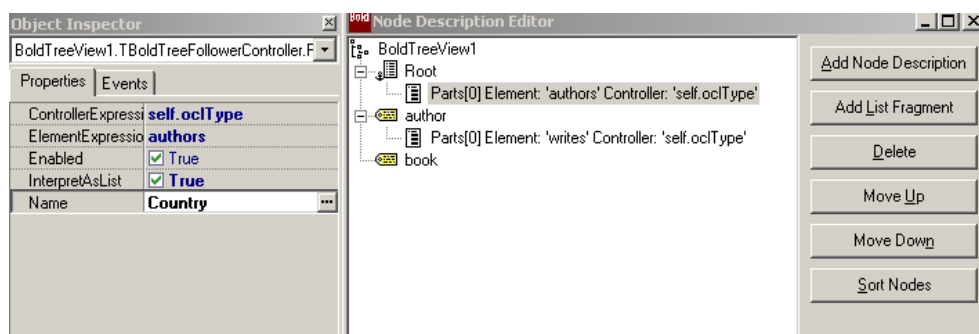


Fig. 9.21. Properties adjustment for hierarchy layer

- ❑ We shall set “self.oclType” OCL-expression to “ControllerExpression” property; the given property is used for searching descriptions of hierarchy nodes at each layer. “Self.oclType” expression sets the current context (see Chapter 5) for all types of the objects included in the given layer, in this case this is “Author” type, in result the component will search all elements of the given type to display at this layer of hierarchy;
- ❑ We shall set to “ElementExpression” property value of association role, linking “Country” and “Authors” classes (see Fig. 9.17), this is “authors” value, i.e. the element of this layer will contain all authors;
- ❑ We shall set “True” value to “InterpretAsList” property, as further we wish to add one more layer of hierarchy including the books of each author;
- ❑ We shall set “Country” string to “Name” property, as the context of the given layer should coincide with a context of the description of a layer element (“ElementExpression” property).

Now we shall add node of hierarchy displaying authors. For this purpose we shall press the button with “AddNodeDescription” name, and set new node as follows (see Fig. 9.22).

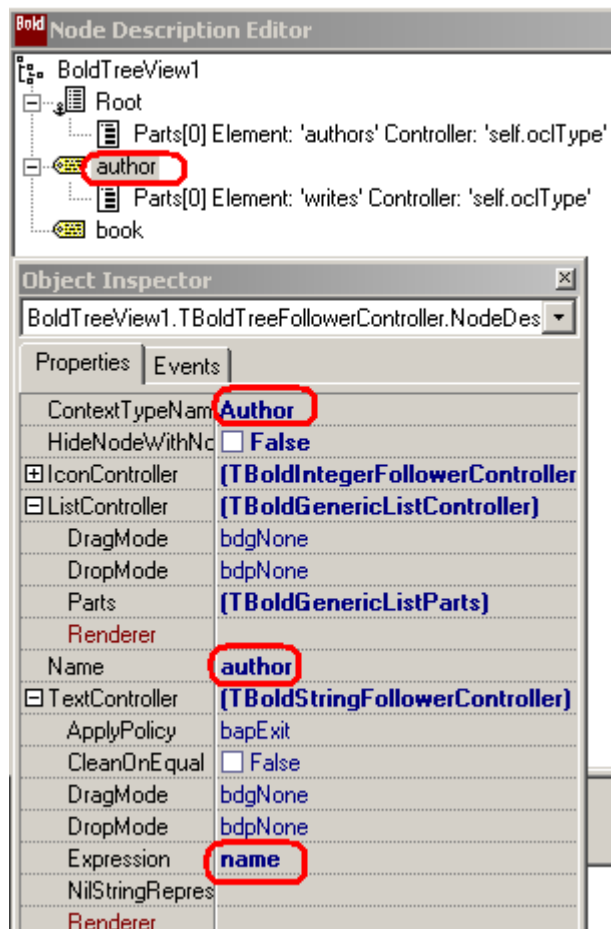


Fig. 9.22. Setting of node properties

- ❑ We shall set “Author” value to “ContextTypeName” property; this property specifies the element type;
- ❑ We shall set “name” OCL-expression to “Expression” property, i.e. we wish to display authors’ surnames.

Let's start the application, and we shall make sure that we have coped with the first part of the task – to display the authors living in the given country.

For creating the second layer of hierarchy we shall repeat the above described actions, concerning not to a root element, but to new “Author” node. First we shall create the description of the hierarchy second layer, and adjust its properties (see Fig. 9.24.).

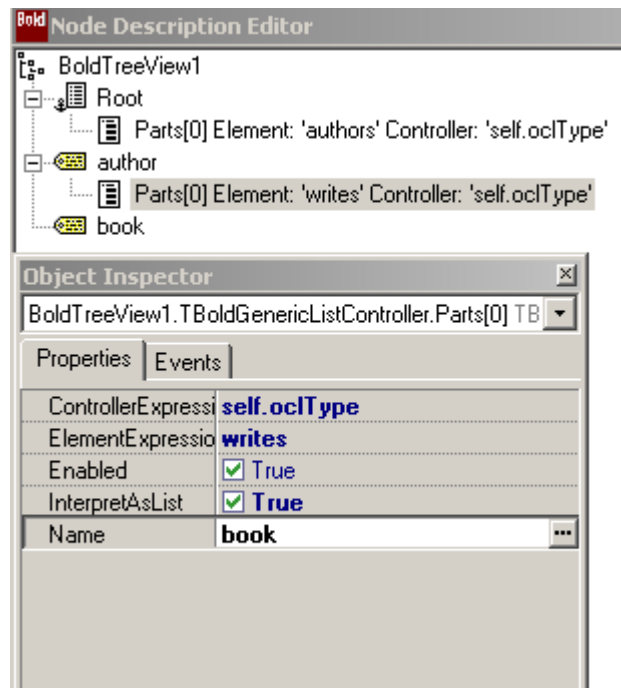


Fig. 9.24. The second layer description adjustment

Then, we shall add new node in the tree to display the list of books, and adjust its properties (see Fig. 9.25).

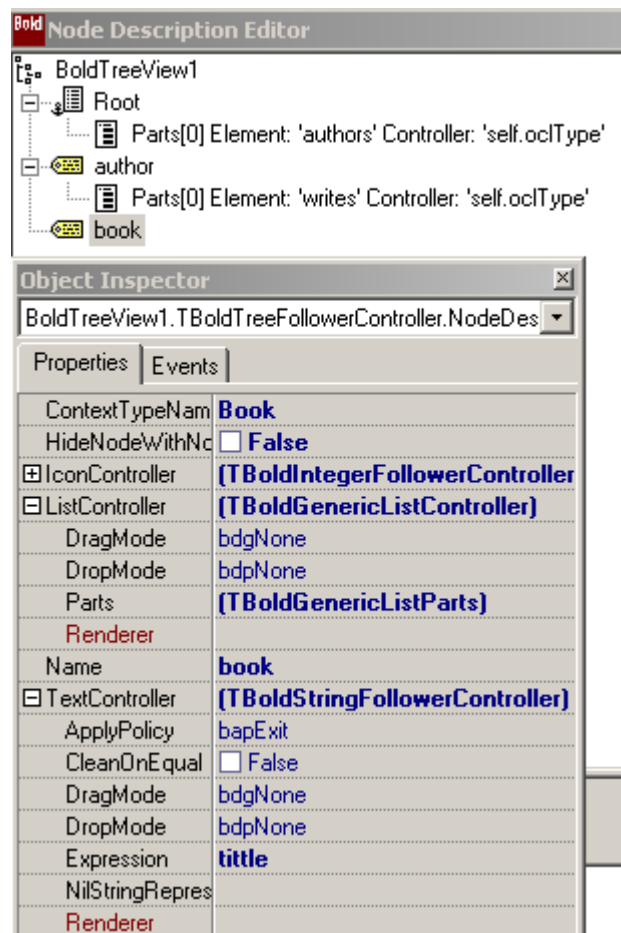


Fig. 9.25. Node properties adjustment for books

In result, having started the application, we shall make sure that the task is solved completely (see Fig. 9.26).

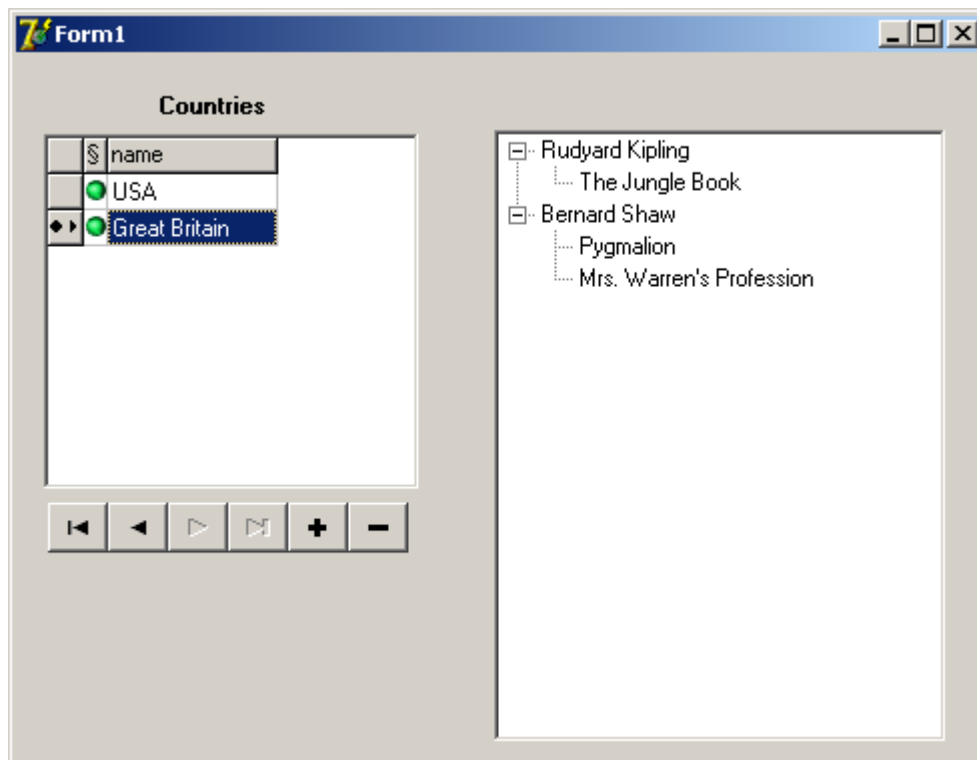


Fig. 9.26. The application in work

It is necessary to note that in the above-stated example only small part of BoldTreeView component capabilities is shown. Actually this component provides much more capabilities, for example:

- ❑ Display of graphic elements (icons) for hierarchy elements;
- ❑ Display of elements, which are not included in object space (created during application run);
- ❑ Capability of describing several layers for one node;
- ❑ Capability of creating a great number of hierarchy layers.

As a matter of fact, the description of all such capabilities would need the separate large chapter. More detailed description of the given component capabilities is represented in the accompanying documentation for Bold for Delphi product, and also is illustrated in two demonstration programs attached to the product provided along with source texts.

BoldImage

The given component is intended for the graphic information display. Such attributes as BLOB, TypedBLOB, BlobImageBMP, BlobImageJpeg, and etc. can contain this information. Use of the given component presents no difficulties; here we shall note only several features.

Loading the Image from File

For loading the image from file into BoldImage component during the program run it is possible to use its LoadFromFile property, as follows:

```
BoldImage1.LoadFromFile(filename);
```

Where filename is the name of file with image.

Using JPEG Format

For supporting work with JPEG-files it is necessary to add two modules into “Uses” section:

```
Uses ..., JPEG, BoldImageJpeg;
```

Stretching Image

The component has “StretchMode” property, which can possess the following values:

- ❑ bsmNoStretch – the image is displayed “as is” (see Fig. 9.27);



Fig. 9.27. bsmNoStretch

- ❑ bsmStretchProportional – the image is scaled proportionally being inscribed by the maximal size in the size of a component (see Fig. 9.28);



Fig. 9.28. bsmStrechProportional

- ❑ bsmStretchToFit– the image possesses the component sizes (see Fig. 9.29);



Fig. 9.29. bsmStretchToFit

- ❑ bsmStretchToScale– the image is scaled proportionally taking into account “Scale” property (see Fig. 9.30).



Fig. 9.30. `bsmStretchToScale` (Scale=50)

Renderers

When describing some visual MDA-components we expressly did not pay attention to their intrinsic common property – `Renderer` property. This property connects to the visual component a special MDA-component with the same name – `Renderer` – that means representative tool. The concept of rendering includes processing some information and its display (representation); it is widely applied, for example, in graphic 3D-packages. In BMDA this concept is similar in many respects, but there under rendering we mean adjusting some properties of the connected visual component - for example, the setting of color, size and type of a font, etc. In other words it is possible to say that renderers serve for representation of the information. Use of renderers allows operating flexibly the specified properties of visual components depending on result of some information processing. And here we have in view an operating control, i.e. the display properties are set at the stage of the application run. As an example we shall consider the following task. We shall assume that we need all USA authors in `BoldGrid` to be colored in red. How can we get it? We shall place on the form `BoldAsStringRenderer1` string renderer from `<BoldControls>` bookmark of Delphi components palette (see Fig. 9.31). The feature of renderers is that they are not connected to visual components. On the contrary, visual MDA-components are connected to renderers. In this case we shall connect to renderer `BoldGrid1` component, or rather its columns. For this purpose we shall double click on `BoldGrid1` and in the called column editor we shall select “name” column. In the objects inspector we shall set `Renderer` property for this column in `BolsAsStringRenderer1` value (see Fig. 9.32). Now it is necessary to specify for the renderer, what information it should process and what properties it should operate. As we posed the task to change color we shall take advantage of `OnSetColor` event, for which we shall write a code of processing according to Listing 9.1.



Fig. 9.31. View of the form with string renderer component

Listing 9.1. Renderer Event Processing for Color Setting

```

procedure TForm1.BoldAsStringRenderer1SetColor(Element: TBoldElement;
  var AColor: TColor; Representation: Integer; Expression: String);
var st:string;
begin
  st:=element.EvaluateExpressionAsString('country.name',1);
  if st='USA' then AColor:=clRed;
end;

```

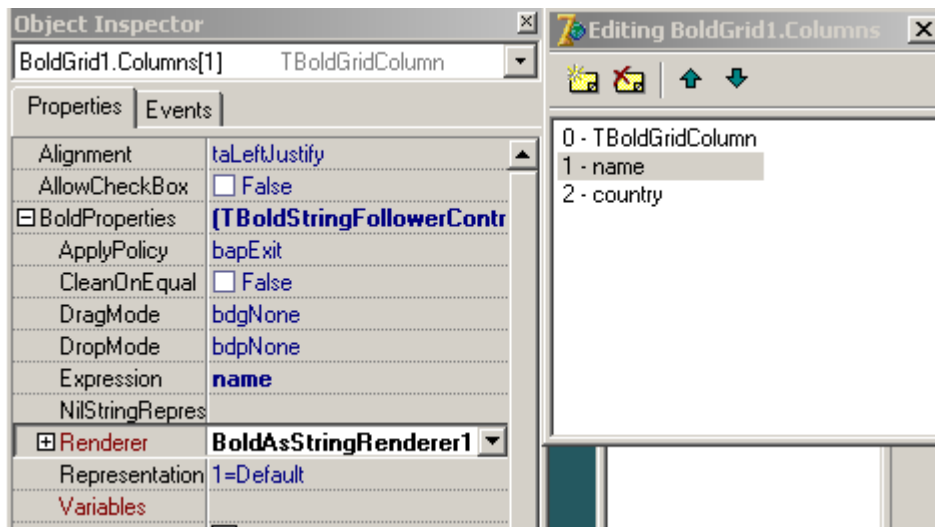


Fig. 9.32. Renderer connecting to BoldGrid column

In the represented code we have used a capability of estimating OCL-expression by means of EvaluateExpressionAsString method of T-BoldElement class. “Country.name” string, which in the context of the selected author returns the name of the country is used here as OCL-expression. After starting the application it is possible to make sure that the task is fulfilled (see Fig. 9.33).

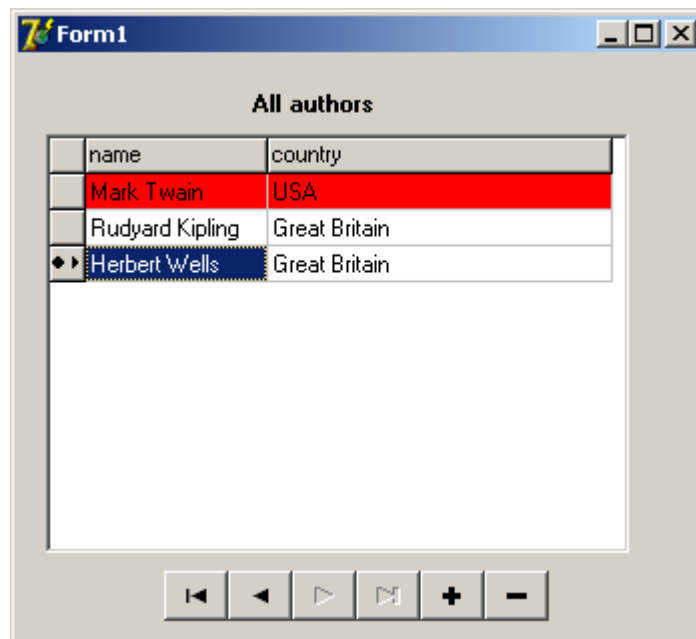


Fig. 9.33. The application run

Similarly it is possible to take advantage of other renderer events, for example, `OnSetFont` (for font parameters setting), `OnGetAsString` - for replacement of the text displayed in the component by any other text depending on set conditions, etc.

Thus, using renderers allows to carry out mapping (representation), i.e. to set rules of display of object space contents on the graphic interface of the application. Taking into account that OCL-expressions estimation methods are included in renderers events handlers, these components basically allow the developer to operate completely the application appearance, setting extremely flexible rules for each visual element of the GUI. We shall note that one renderer can simultaneously “serve” several various visual components that allow creating the certain style.

Autoforms

General Information. Autoform Structure.

Autoforms are special forms automatically generated by BMDA environment for viewing and editing the data supporting Drag-n-Drop interface. We have already dealt with

autoforms, when considered creating the simple application (see Chapter 3). Autoforms are initialized by default at double click on some visual MDA-components. For involving autoforms it is necessary to register “BoldAFPDefault” module name in Uses section of the form module. Use of autoforms allows carrying out navigation on model easily. We shall examine in more detail structure of generated autoforms by the example considered in Chapter 9. When double clicking on the string of a grid (the list of the countries display) with the name of the country – “Russia” – the corresponding autoform appears. By default it looks like the form represented in Fig. 9.34. The heading - the name of the autoform is defined by DefaultStringRepresentation tagged value (see Chapter 4). If it is not specified, BMDA internal index information is displayed.

In the middle part of the autoform the area of attributes is located (it is outlined with a rectangular in Fig. 9.34). In this area names of class attributes are represented (in this case “Country” class has a single attribute with the name – “Name”), and near each name the editing window, which displays the current contents of the given attribute, is located (in our case this is the name of the country – “Russia”).



Fig. 9.34. View of autoform by default

By means of the specified editor window it is possible to change value of any attribute directly on the autoform.

At the top of the autoform the area of multiassociations is located (it is outlined with the rectangular with round corners in Fig. 9.34). This area represents a set of bookmarks, which quantity corresponds to the quantity of associations of the given class + one main bookmark with the class name (in this case this is “Country”). At that all these associations have multiplicity more than 1 (multiassociation).

If we look at UML-model fragment (see Fig. 9.35), we shall make sure that “Country” class has two associations linking it with “Author” and “Publishing house” classes. Thus as it is easy to see, names of bookmarks on the autoform correspond to names of these associations ends – “Author” and “Publishing houses” accordingly. One more bookmark with “History” name is intended for the purposes of tracking the system changes, and now we shall not discuss it. We shall note that for its concealment it is enough to set BoldShowHistoryInAutoForms global variable in false value. It can be made, for example, in section of the module initialization.

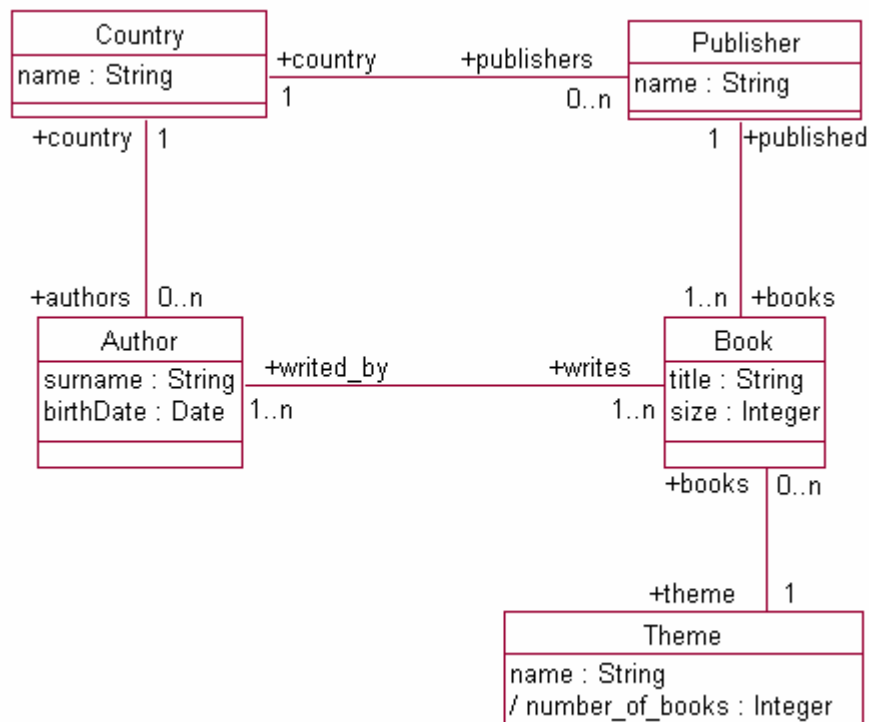


Fig. 9.35. UML-module fragment

If we select, for example, “Authors” bookmark on the autoform the window of the corresponding association will open. In this window all objects of the “Authors” linked class (see Fig. 9.36) in the available for editing grid of BoldGrid type will be displayed; and also the Bold-navigator, with which help it is possible to delete or add objects such as “Author”, is displayed.

Autoforms automatically possess property to generate derived autoforms when double clicking by the mouse button. So, if we click on the autoform grid string with “Ilf” author name, the derived autoform, which contains the information on this author, will be displayed (see Fig. 9.37). We could receive it in another way - directly having double cluck on the corresponding grid string of the main application form displaying the list of all authors.

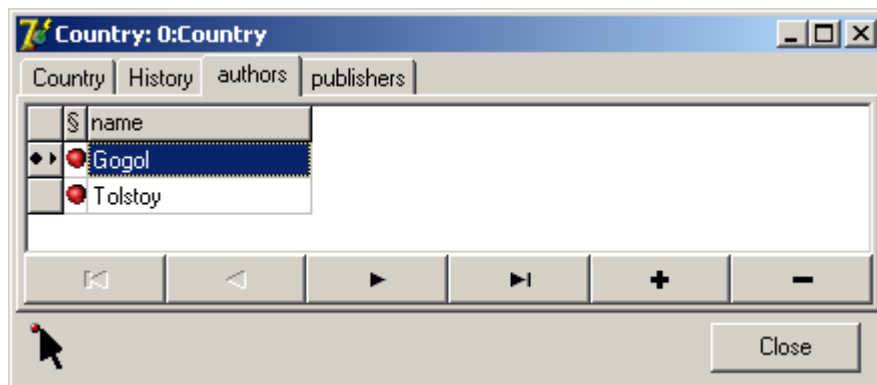


Fig. 9.36. View of the autoform with multiassociation window

Let's note that the autoform of the author differs from the autoform of the country by presence of a grey field with "Russia" country name (see Fig. 9.37), which does not allow direct editing. Such grey fields display single association of the given class (in model the rule is incorporated: the author can live only in one country).

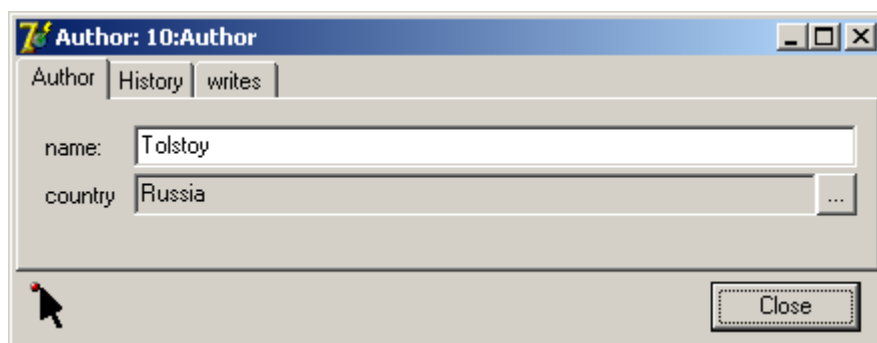


Fig. 9.37. View of the derived autoform with single association

For editing associations in autoforms Drag-n-Drop interface is actively used. Any autoform contains "a point of link" (it is outlined with a circle in Fig. 9.34), displayed as an arrow with a red point. If we "drag" the autoform represented in Fig. 9.34 by this point with the help of the mouse, move it onto the autoform shown in Fig. 9.37, and release the mouse button above a grey field with the name of the country, in this way we will link the concrete country to the concrete author, at that in a grey field the name of the new country will be displayed. In such a way it is possible "to drag" autoforms and "to release" them on grids inside other autoforms realizing multiassociations. We shall remind that it is possible "to drag" directly BoldGrid string of the usual form on the autoform, as we did it in Chapter 3.

Besides usability all these capabilities also provide consistency of the data as direct editing of such grey fields for single associations is forbidden, i.e. it is possible to set only existing objects. On the contrary, for multiassociations it is possible to add objects of the linked class directly on the autoform. So, if we, taking advantage of the navigator on the autoform represented in Fig. 9.36, shall add new, not existing up to this moment, authors (though this autoform is linked not to “Author” class, but to “Country” class!) we shall carry out such operation without difficulty, but at that these new objects will be added automatically into “Author” class.

Constraints

Autoforms have a number of constraints. They have no visual elements for display of attributes such as Memo, attributes such as images or BLOB-attributes. Autoforms are not adjusted in the ordinary way, i.e. we cannot place pull-down lists, buttons, etc. on them.

NOTE

Individual adjustment of autoforms basically is possible only at the layer of the source text. For this purpose it is possible to take advantage of the delivered source text of BoldAFPDefault.pas program module.

Therefore, from the point of view of practice, autoforms are expedient for using for object space contents scan with the purpose of debugging, or for editing simple text or numerical attributes.

Generating Autoforms

In case of need the developer can initiate generation of the autoform independently, for example, by pressing the button. For this purpose `AutoFormProviderRegistry` special class is used. In Listing 9.2 simple procedure, which can be used for generating the autoform directly from the application program code, is presented. For its use it is necessary to add BoldAFP module into Uses section.

Listing 9.2. Generating Autoform by Program Means

```
procedure ShowAutoFormForClass(ClassName:String);
```



```

var F : TForm;
F
:=AutoFormProviderRegistry.FormForElement(BoldSystemHandle.System.ClassByExp
ressionName[(ClassName)]);
  if Assigned(F) then F.Show;
end;

```

Here BoldSystemHandle is the system handle, and ClassName is model class name, for which it is necessary to generate autoform.

Attributes Display Control

As it has been already told, the autoform is not capable to display attributes of the certain types. If not to undertake any actions, nevertheless, display of such attributes on the autoform will always occur, but in this case all these attributes will be displayed in usual single-string editing text fields. At that, for example, for the attribute containing a graphic image, or the RTF-text (BLOB-attributes), contents of the specified editing windows can look rather strange and to be not readable. With the purpose of managing structure of attributes displayed on autoforms it is possible to take advantage of BoldPlaceableAFP component from BoldMisc bookmark. This component supports OnMemberShouldBeDisplayed event, in which handler it is possible to place a program code for forming condition of concrete attribute display. The example of such code is represented in Listing 9.3.

If we place the specified component on the main form, and enter the given processing code we shall see that the name of the author will not be displayed on autoforms . In this case we check by program, whether the name of the current attribute coincides with “Author.name” expression, and, if it is so, we shall set ShowMember attribute of display in False value (see Listing 9.3).

Listing 9.3. Attribute Display Control

```

procedure TForm1.BoldPlaceableAFP1MemberShouldBeDisplayed(
  Member: TBoldMemberRTInfo; var ShowMember: Boolean);
var st:string;
begin
  st:=member.EvaluateExpressionAsString('self',1);
  if st='Author.name' then ShowMember:=false;
end;

```

Additional Tools

BoldCaptionController

The given component is intended for automatic formation of the visual name of any VCL-component (“Caption” property) depending on OCL-expression value specified in its “Expression” property. We shall place BoldCaptionController component on the form, and link it to the handle of the countries list from the example examined earlier (see Fig. 9.39). Also we shall add Panel1 – usual VCL-component – on the form, and specify it in “TrackControl” property of BoldCaptionController component (see Fig. 9.39).

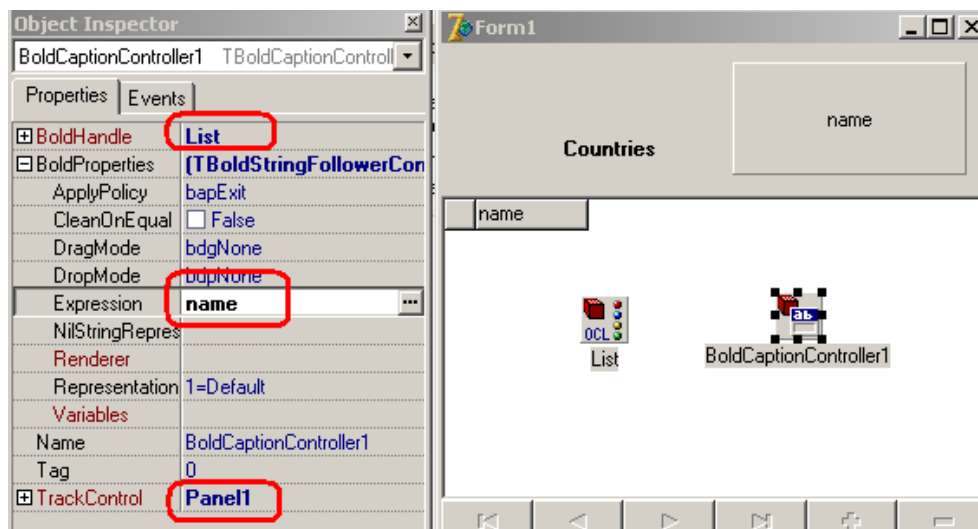


Fig. 9.39. Adjustment of BoldCaptionController

Let's start the application and we shall make sure that when moving in the list of the countries on the panel the name of the current country is displayed .

Summary

In this chapter we have got acquainted with some visual components of Bold for Delphi environment. From the considered examples it is clear that the main principle and the purposes of such components creation is the maximal automation of behaviour of the application graphic interface in accordance with business-layer set by handles of object space.

Chapter 10. Work with Persistence Layer

The possibility of effective work with the data is one of the key functionalities given by Bold for Delphi environment. At that Bold is not limited only to interoperability with the DBMS, but provides convenient means for automatic generation of databases for application UML-model. Besides the structure of Bold for Delphi tools arsenal includes effective means for saving objects in XML-documents. In this chapter we shall familiarize with Bold for Delphi basic components, intended for work with the data, and also we shall consider by numerous examples issues of practical implementation of the basic capabilities given by Bold for Delphi for work with relational DBMS and XML-documents.

Persistence Layer Functions

The persistence layer is intended for providing realization of the following basic functions:

- ❑ Saving the elements (objects and associations) of object space (OS) in long-term memory (usually on hard disk, though for this purpose any other media devices - diskettes, flash-disks, CD / DVD-RW, etc. can be used in the presence of standard access to them from the application). Thus only those objects and links can be saved, for which Persistent property is set in model of the application;
- ❑ Loading the elements of object space from long-term memory;
- ❑ Supporting the mechanism of object-relational mapping, i.e. transformation of object UML-model into structure of a relational database;
- ❑ Generating the scheme of the relational database on the available object model;
- ❑ Transformation of OCL-expressions into SQL operators (so-called “OCL2SQL”).

Components Structure

In Fig. 10.1 the scheme illustrating interoperability of object space and persistence layer is presented. The persistence layer descriptor is the central element organizing such interoperability. Both BoldPersistenceHandleFileXML component intended for saving the OS in XML-file (we have already used this component when describing creation of the simple application (see Chapter 3), and BoldPersistenceHandleDB component intended for the organization of interoperability with relational DBMS can act as such descriptor.

For supporting work with concrete types of the DBMS special components-adapters of databases are included into Bold for Delphi structure (see Fig. 10.1).

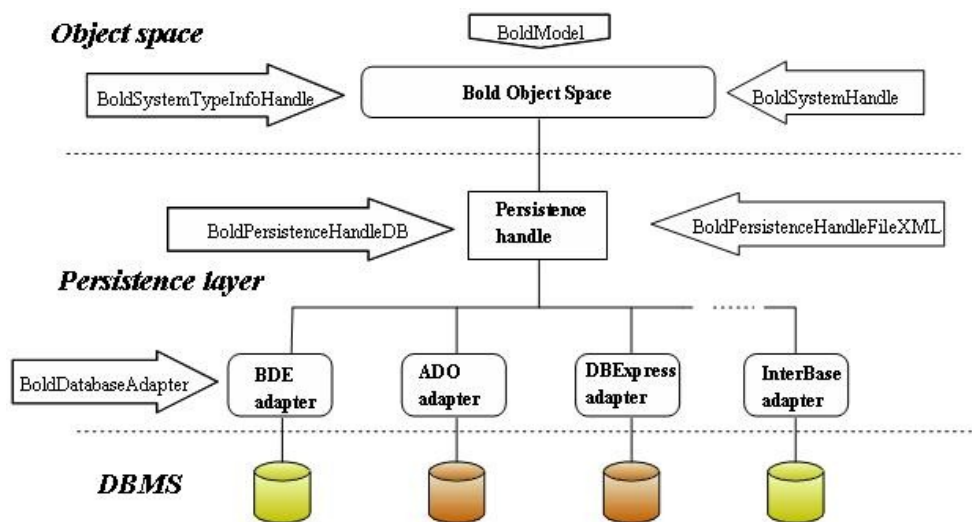


Fig. 10.1. Structure of the persistence layer and its links

There are following basic databases adapters in the considered Borland MDA version:

- ❑ BoldDataBaseAdapterBDE — provides connection to the DBMS through Borland Database Engine;
- ❑ BoldDataBaseAdapterADO — provides connection to the DBMS through ActiveX Data Objects (ADO) interface;
- ❑ BoldDataBaseAdapterIB — provides connection to Interbase DBMS;
- ❑ BoldDataBaseAdapterDBX — provides connection to the DBMS through DBExpress interface.

Besides the adapters listed above, there are adapters for the organization of interoperability with the data by means of SOAP (Simple Object Access Protocol), and also for DBISAM DBMS and for SQLDirect package. However, as it is shown below, the developer has an opportunity to create his own DBMS adapter if necessary.

Work with DBMS

Persistence Layer Connection

Let's consider, how in practice connection to the DBMS is carried out. For this purpose the application created earlier (see Chapter 8) consisting of one form and the module of the data is used. A model of the application (see Fig. 10.2) we shall leave without changes.

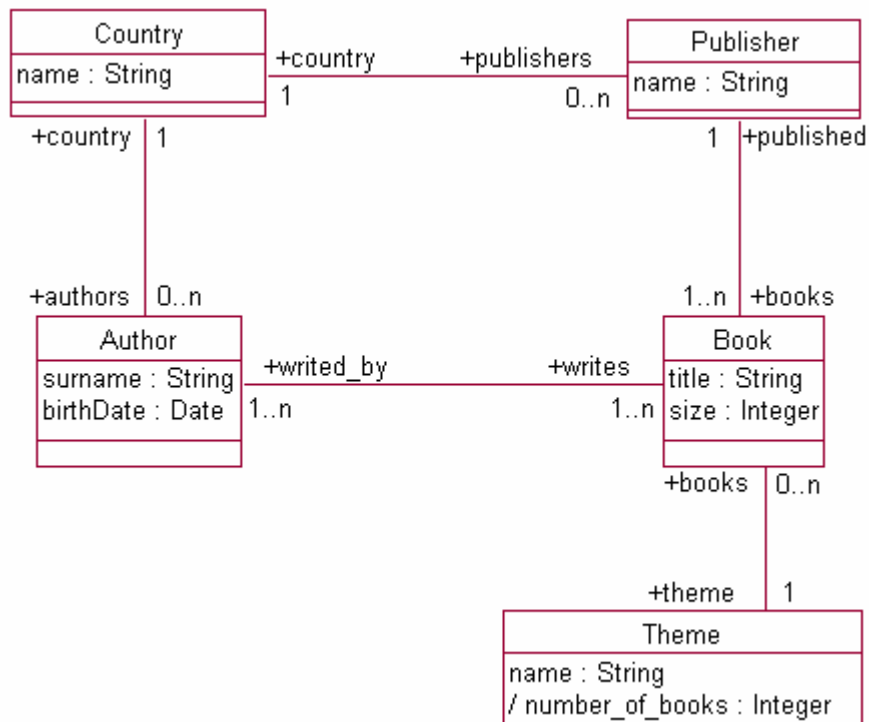


Fig. 10.2. The using model

Let's remove from the data module `BoldPersistenceHandleFileXML` component used earlier, and instead of it we shall add `BoldPersistenceHandleDB1`, `BoldDatabaseAdapterIB1` components from `BoldPersistence` bookmark, and also a component, which represents Interbase - `IBDatabase1` (see Fig. 10.3).

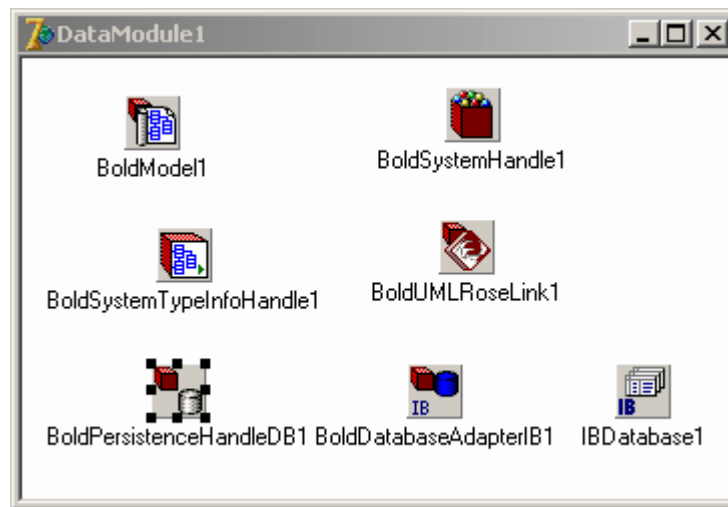


Fig. 10.3. Data module components structure

Let's adjust the added components as follows (see Fig. 10.4).

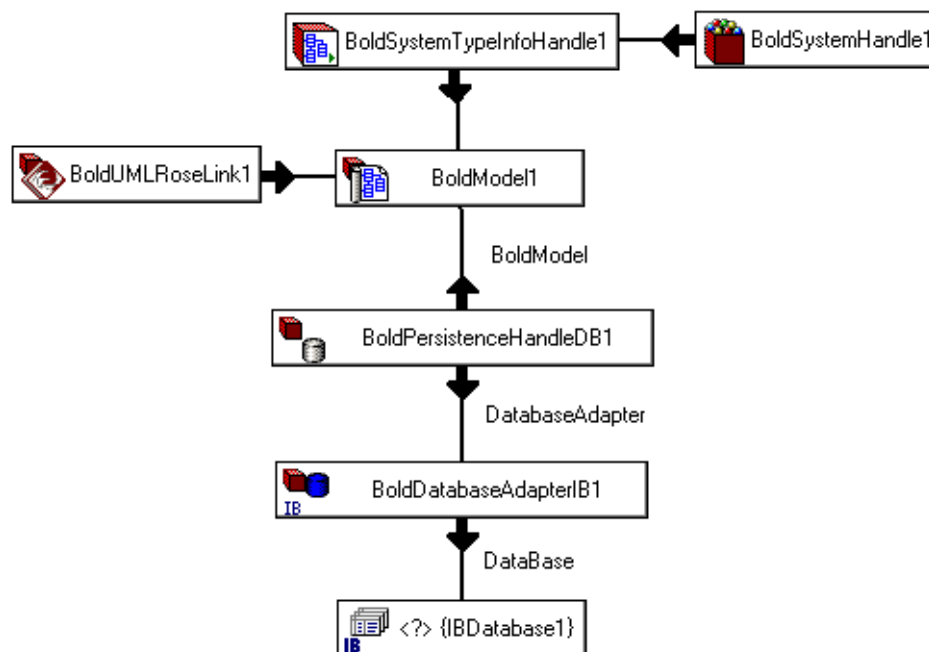


Fig. 10.4. Diagram of data module properties adjustment

Let's create Interbase empty database with the help of IBConsole standard application. We shall give it lib.gdb name and specify its location, for example, in disk C root: (see Fig. 10.5).

The 'Create Database' dialog box is shown with the following details:

- Server:** Local Server
- File(s):**

Filename(s)	Size (Pages)
C:\LIB.GDB	400
- Options:**

Page Size	4096
Default Character Set	WIN1251
SQL Dialect	1
- ☒ Register database
- Alias:** LIB
- Buttons: OK, Cancel

Fig. 10.5. Creating the empty database

Let's connect IBDatabase1 component to the created database, and set 1 value to its SQLDialect property. Then we shall check up connection with the DB (see Fig. 10.6).

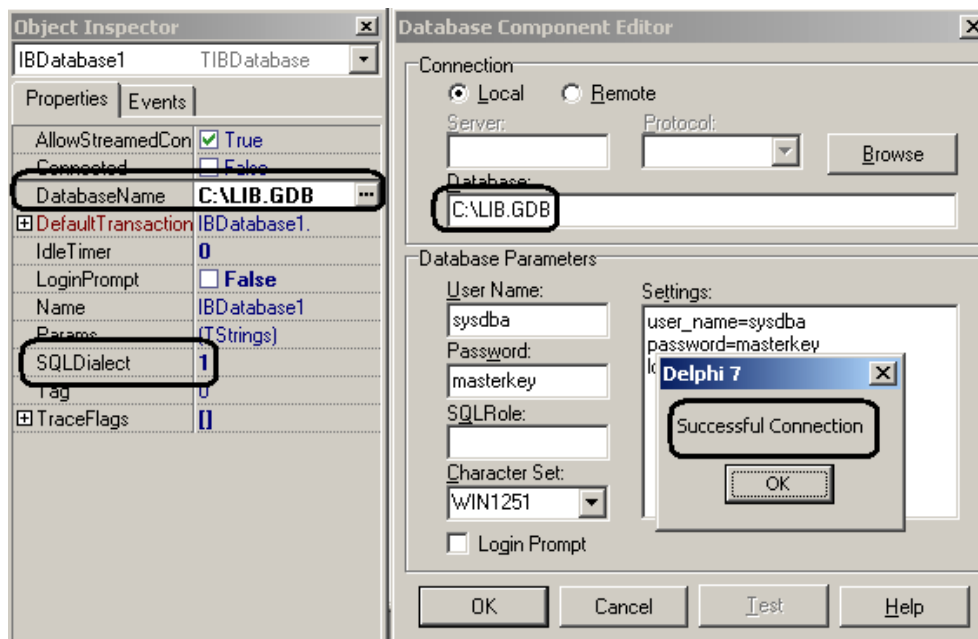


Fig. 10.6. Adjustment and verification of connection with the database

Further, in the object inspector we shall adjust the database adapter, in this case having specified simultaneously the type of DBMS and used SQL-dialect by means of selecting “dbInterbaseSQLDialect1” item of “DatabaseEngine” property from the pull-down list (see Fig. 10.7).

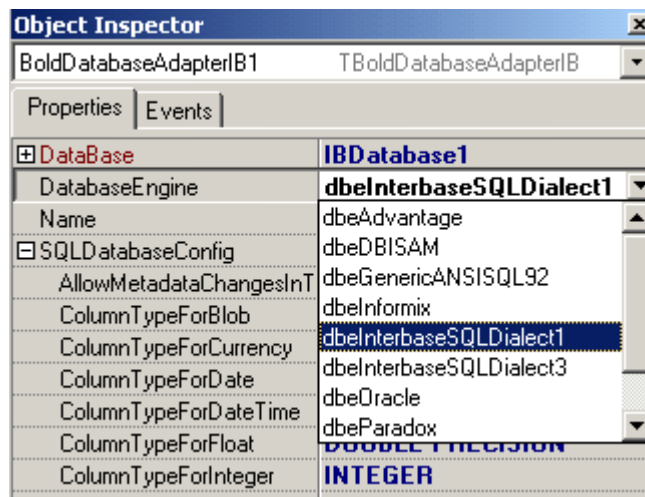


Fig. 10.7. Adjustment of DBMS adapter

Generating Database Scheme

For generating database we shall start the editor of models and choose Tools ► Generate Database subitem from the main menu or simply press the button of the top toolbar with the DB image (see Fig. 10.8).

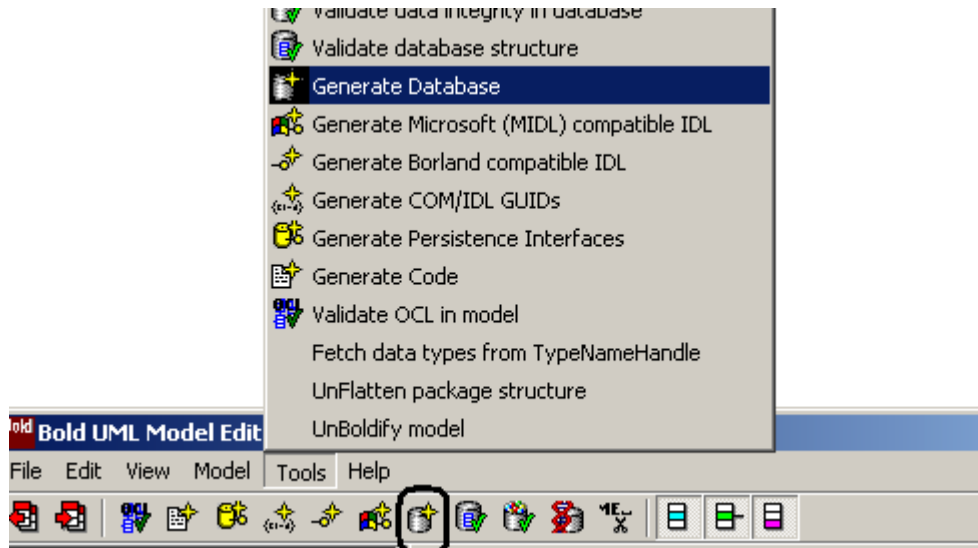


Fig. 10.8. Ways of DB generation start

The following window with the enquiry message for confirmation will appear (see Fig. 10.9).

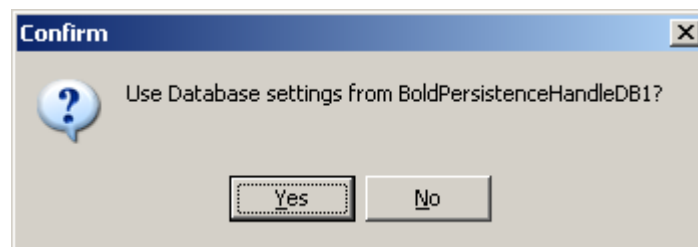


Fig. 10.9. The window with enquiry message of environment

The given window requests confirmation on use of BoldPersistenceHandleDB1 component for generating the DB scheme. If our application contained several such components, then after the answer – “No” we would receive consistently the same windows with enquiries- confirmations of using other similar components. In this case we shall simply press “Yes” button. Further the behaviour of environment depends on database – whether it is empty, or it is already filled with some information. If the database already contains the information,

the warning window will be displayed (see Fig. 10.10), informing that all data in DB will be lost as a result of structure generation.

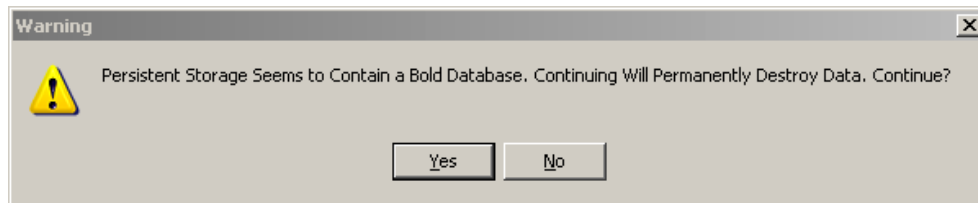


Fig. 10.10. Probable data loss warning

In our case the database is empty yet, therefore Bold environment will start generating the DB scheme according to our UML-model right away. The course of the scheme generation can be traced by the information in the status bar of the editor. Time spent on procedure of DB structure generation depends on number of model elements and from computer processing speed. For our model it makes about 2-3 seconds. After the DB generation procedure completion we can use IBEasy utility , and see, what tables of a database has Bold generated.

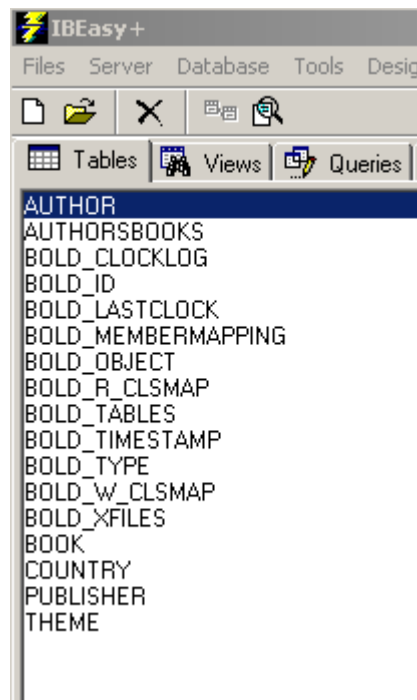


Fig. 10.11. Structure of generated DB tables

System Tables Structure

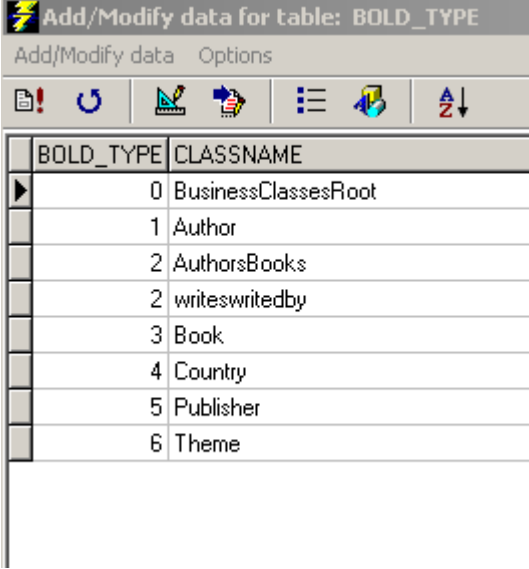
As can be seen in Fig. 10.11, there are some tables, which we “did not reserve” in structure of the tables automatically generated by Bold for Delphi environment. They were automatically created with environment for its own use. Names of such *system* tables begin with “Bold_” prefix (see Fig. 10.11).

NOTE

By request the developer can specify a prefix added by Bold for Delphi environment to system tables names independently. For this purpose it is enough to set a necessary prefix in string property of the DBMS component-adapter (in the considered example it is BoldDatabaseAdapterIB).

Let's consider structure and purpose of system tables.

- ❑ Bold_ID – the given table always contains the single string containing one field – ID integer, used by the environment for assignment to new objects at their first retention in the database; initial value of this field is equal to 1;
- ❑ Bold_TYPE – this table sets and holds conformity between a concrete class of model and the integer determining its type; for example, the given table contains the following rules of types mapping (see Fig. 10.12);



BOLD_TYPE	CLASSNAME
0	BusinessClassesRoot
1	Author
2	AuthorsBooks
2	writeswritedby
3	Book
4	Country
5	Publisher
6	Theme

Fig. 10.12. Bold_TYPE–mapping of classes types

- ❑ **Bold_TABLES** – names of all DB tables (including system) are kept in this table databases; the information from this table is used by environment for the analyzing the conditions of safe removal of tables at updating structure of the database;
- ❑ **Bold_TIMESTAMP** – contains a single whole field, which value increases by 1 at each call of “UpdateDatabase” method (i.e. at each synchronization of object space and persistence layer). Initial value of the given field is equal to 0; it is possible to permit or forbid use of the given table by setting the global property of model – “UseTimeStamp” (see Chapter 4);
- ❑ **Bold_XFILES** – the given table keeps the whole history of the database. When creating any object its global unique identifier (GUID) is entered into the given table and is never deleted from it, even if the object has been removed from a database. The table is used at synchronization of the information with external databases. Besides GUID of each object, the given table stores the time marker corresponding to the moment of its last change. It is possible to permit or forbid use of the given table by setting the global property of model – “UseXFiles” (see Chapter 4). In the given table it is possible to permit or forbid use of the time marker by setting the global property of model – “UseTimeStamp” (see Chapter 4);
- ❑ **Bold_CLOCKLOG** – it is intended for binding integer values of TimeStamp time markers to physical values of date and time. At great volume of frequent changes of a database it is possible to set ClockLogGranularity parameter managing time refinement of the nearby recordings. It is possible to permit or forbid use of the given table by setting global property of model – “UseClockLog” (see Chapter 4);
- ❑ **Bold_LASTCLOCK** – it is used jointly with the previous table for storing the time moment of entering the last recording in it;

Other system tables are intended for realizing the mechanism of “database evolution”, and they will be considered later.

The Generated Tables Structure

Let's examine in brief how does Bold for Delphi environment transform UML-model elements (classes, attributes, associations) into tables and fields of a database. By means of IBEasy utility we shall obtain structure of “Author” table (see Fig. 10.13).

Mnemonic	Domain	Description
BOLD_ID	INTEGER	
BOLD_TYPE	SMALLINT	
NAME	VARCHAR	
COUNTRY	INTEGER	

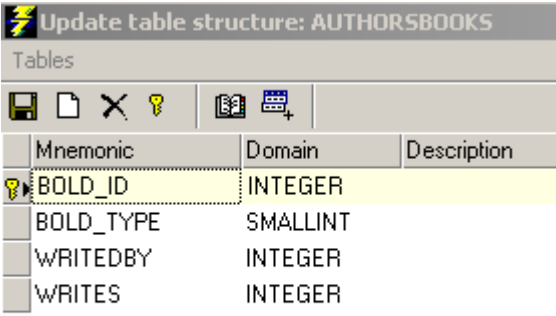
Fig. 10.13. “Author” table structure

We shall consider the fields present in the given table. “Bold_ID” field is a primary autoincremental key of the table. “Bold_Type” field determines type of recording using “Bold_Type” system table described above for types mapping.

ATTENTION

Classes-heirs and inheritance relations were not entered into the considered model intentionally, with the purpose of simplification of the material statement. For such simple cases “Bold_Type” field will always contain identical values within the framework of one table. For more difficult situations, when the given table reflects the class-parent (the model super class), this field will contain various values, which will coincide with types of descendants of the mapped super class.

“Name” field is usual string attribute with the empty string set by default. In this case “COUNTRY” field is a key specifying recordings of “COUNTRY” master - table of the same name, as “one-to-many” relation connects «Country» and «Author» classes of the model. Further, we shall consider one more table “AUTHORSBOOKS” (see Fig. 10.14).



Mnemonic	Domain	Description
BOLD_ID	INTEGER	
BOLD_TYPE	SMALLINT	
WRITEDBY	INTEGER	
WRITES	INTEGER	

Fig. 10.14. Binding table structure

The structure of this table also is simple enough. The given table as it is easy to understand, is the binding table, and it is intended for storage of the information on link, such as "many-to-many", which reflects presence of association in UML-models between "Author" and "Book" classes. It is obvious that the names of fields used for links storage ("WRITEDBY" и "WRITES") are names of roles of this association.

Thus, the structure of tables generated by Bold for Delphi environment, is transparent and understandable enough. Basically, situations are possible, when there is a necessity to use a database, which scheme has been generated by Bold environment, from external applications, which have been developed by traditional way. Taking into account the examples considered above, it is possible to draw a conclusion that such task is quite solvable.

Using DBMS Adapters

In the example considered above we used BoldDataBaseAdapterIB component providing work with Interbase DBMS. In Bold for Delphi arsenal there is a set of the similar components intended for connection to other types of databases (BoldDataBaseAdapterBDE, BoldDatabaseAdapterADO, etc.). All these components are on "BoldPersistence" bookmark of Delphi components palette. Use of the specified adapters is practically similar to the considered adapter of Interbase DBMS, thus it is necessary to replace IBDatabase component with corresponding analogues only. For example, when using Microsoft SQL Server DBMS it is logical to use TDataBaseAdapterAdo component together with ADOConnection. All adapters of the DBMS have the adjustable interface of SQL dialect (see Fig. 10.16), determined by "SQLDataBaseConfig" property. The given property is complex, and it contains set of concrete subproperties-settings allowing adapting the component-adapter for the concrete DBMS or its clone. For example, some DBMS have constraints on length of the identifier of the string field, thus for specifying the concrete constraint it is possible to use "DefaultStringLength" property. In some cases use of "UseSQL92Joins" property allows to

increase productivity of work with DBMS (naturally if the given DBMS supports such queries-connections). These queries have the following syntax:

```
SELECT <Field name> FROM <Table_name> <Table_alias>
LEFT JOIN < Table_name > < Table_alias > ON <Primary_key>
WHERE <Fetching_condition>
```

The sense of other adjustments is quite clear from their names, and we shall not stop in detail on them.

SQLDatabaseConfig	(T-BoldSQLDataBaseConfig)
AllowMetadataChangesInTransact	<input checked="" type="checkbox"/> True
ColumnTypeForBlob	BLOB
ColumnTypeForCurrency	DOUBLE PRECISION
ColumnTypeForDate	DATE
ColumnTypeForDateTime	DATE
ColumnTypeForFloat	DOUBLE PRECISION
ColumnTypeForInteger	INTEGER
ColumnTypeForSmallInt	SMALLINT
ColumnTypeForString	VARCHAR(%d)
ColumnTypeForTime	DATE
DBGenerationMode	dbgQuery
DefaultStringLength	255
DropColumnTemplate	ALTER TABLE <TableName>
DropIndexTemplate	DROP INDEX <IndexName>
DropTableTemplate	DROP TABLE <TableName>
EmptyStringMarker	
FetchBlockSize	250
FieldTypeForBlob	ftBlob
MaxDbIdentifierLength	31
MaxIndexNameLength	31
MaxParamsInIdList	20
QuoteNonStringDefaultValues	<input type="checkbox"/> False
ReservedWords	(TStringList)
SQLforNotNull	NOT NULL
StoreEmptyStringsAsNULL	<input type="checkbox"/> False
SupportsConstraintsInCreateTable	<input checked="" type="checkbox"/> True
SupportsStringDefaultValues	<input checked="" type="checkbox"/> True
SystemTablePrefix	BOLD
UseSQL92Joins	<input type="checkbox"/> False

Fig. 10.16. Adjusting SQL dialect

Creating DBMS Own Adapters

In spite of a wide spectrum of interfaces of interoperability with DBMS covered by Bold for Delphi environment (BDE, ADO, DBExpress, IBExpress, etc.), which allow to work with the overwhelming majority SQL-servers and local DBMS existing at present, nevertheless, it is impossible to exclude a situation, when the developer should solve the task to provide the functioning of the application with some “unique” database, which does not support the above-stated interfaces. In that case the task can be solved, having created the own component – the adapter for the given concrete DBMS. However, before this it is useful to familiarize with requirements and constraints, which Bold for Delphi environment demands from databases control systems.

DBMS Requirements

Not every DBMS can be used jointly with Bold for Delphi. For such interoperability performance of three basic conditions is necessary. First, the DBMS should be relational database.

NOTE

Probably, Bold for Delphi developers, have acted absolutely correctly, having developed the mechanism of object-relational mapping for transforming the object-oriented structure (UML-models) of classes diagrams into the "foreign" relational data model, though they had to overcome many difficulties on this way. Such situation can be explained by the fact that, most likely, relational DBMS will dominant for a long time over object-oriented databases (OODB) in respect to distribution and availability. For today there are not many perfect industrial OODB, there are only a few. Though, certainly, when using OODB, volume and complexity (and hence the cost) of such software product as Bold for Delphi, would cardinally decrease.

Second, the DBMS should support language of data definition – DDL (Data Definition Language), for manipulating objects of a database (creation, change, etc.). The matter is that for generating the DB scheme Bold always generates special SQL-script, which, actually, realizes generation of the database scheme. This script always uses DDL-operators. For an illustration we shall show a small fragment (see Listing 10.1) of SQL-script, which is generated by Bold environment for creating the scheme of a database in the example considered above.

Listing 10.1. Fragment of SQL-script for database scheme generation

```
CREATE TABLE BOLD_ID ( BOLD_ID INTEGER NOT NULL)
CREATE TABLE BOLD_TYPE ( BOLD_TYPE SMALLINT NOT NULL,
    CLASSNAME VARCHAR(255) NOT NULL)
CREATE TABLE BOLD_XFILES ( BOLD_ID INTEGER NOT NULL,
    BOLD_TYPE SMALLINT NOT NULL,
    EXTERNAL_ID VARCHAR(255) NOT NULL,
    BOLD_TIME_STAMP INTEGER NOT NULL,
```



```

        CONSTRAINT IX_BOLD_XFILES_BOLD_ID PRIMARY KEY (BOLD_ID))
CREATE TABLE BOLD_TABLES (  TABLENAME VARCHAR(255)  NOT NULL)
CREATE TABLE BOLD_TIMESTAMP (  BOLD_TIME_STAMP INTEGER  NOT NULL)
CREATE TABLE BOLD_LASTCLOCK (  LastTimestamp INTEGER  NOT NULL,
    LastClockTime DATE  NOT NULL)
CREATE TABLE BOLD_CLOCKLOG (  LastTimestamp INTEGER  NOT NULL,
    ThisTimestamp INTEGER  NOT NULL,
    LastClockTime DATE  NOT NULL,
    ThisClockTime DATE  NOT NULL)
CREATE TABLE BOLD_MEMBERMAPPING (  CLASSNAME VARCHAR(60)  NOT NULL,
    MEMBERNAME VARCHAR(60)  NOT NULL,
    TABLENAME VARCHAR(60)  NOT NULL,
    COLUMNS VARCHAR(60)  NOT NULL,
    MAPPERNAME VARCHAR(60)  NOT NULL)
CREATE TABLE BOLD_R_CLSMAP (  CLASSNAME VARCHAR(60)  NOT NULL,
    TABLENAME VARCHAR(60)  NOT NULL,

```

We can get the text of such script, having used the following operator (see Listing 10.2).

Listing 10.2. The operator of SQL-script for generating the DB scheme

```

datamodule1.BoldPersistenceHandledB1.
PersistenceControllerDefault.PersistenceMapper.GenerateDatabaseScript(
memo1.Lines, '');

```

In the given listing the text of a script is located in the string file of Memo1 component of TMemo type though for this purpose it is possible to use any class of TStrings type.

And third, the DBMS should support transactions, and their management (StartTransaction, CommitTransaction, RollbackTransaction commands) as Bold executive system should provide the control and management of modification process in the persistence layer.

Program Modules Structure of DBMS Adapter

Any DBMS component-adapter used by Bold environment consist of the following basic modules (by the example of ADO adapter):

- ❑ BoldDatabaseAdapterAdo.pas – provides connection with a component representing the DBMS (in this case – with ADOConnection). It forms Bold internal variable describing a database – InternalDatabase;
- ❑ BoldPersistenceHandleADO.pas – sets connection with the database;
- ❑ BoldPersistenceHandleADOREg.pas – logs components in IDE;
- ❑ BoldADOInterfaces.pas – realizes all operations with DBMS, i.e. generation of queries, transactions management, etc.

All sources of listed above program modules-components of DBMS adapters are provided with Bold for Delphi product. Having used them as examples and analogues, the developer has an opportunity to create own DBMS adapter.

Adapters Use Advantages

The described scheme of connection to the DBMS and capabilities of DB scheme generation provides the following qualitative advantage when creating databases applications. If during the application development a necessity occurs of the created application "translation", for example, from local MS Access DBMS to client-server Oracle DBMS, then in this case at traditional development it is necessary to create, most likely, a database in Oracle practically anew. In case of Bold for Delphi use it is enough to connect another DBMS adapter and to generate automatically structure of a new database.

BoldActions Use for Work with DB

Structure Borland MDA includes convenient enough means for providing automation of actions on databases creation, and management of connection to a persistence layer at the stage of the application execution – BoldActions.

NOTE

In considered version and existing for today releases of Bold for Delphi use of these means is possible only when working with Interbase DBMS.

The involvement of these tools is carried out by means of standard Delphi-component - ActionList. We shall place it on our form, then call by the mouse right button the pop-up menu, and select NewStandartAction item. In the opened window (see Fig. 10.17) we shall select TBoldIBDatabaseAction.

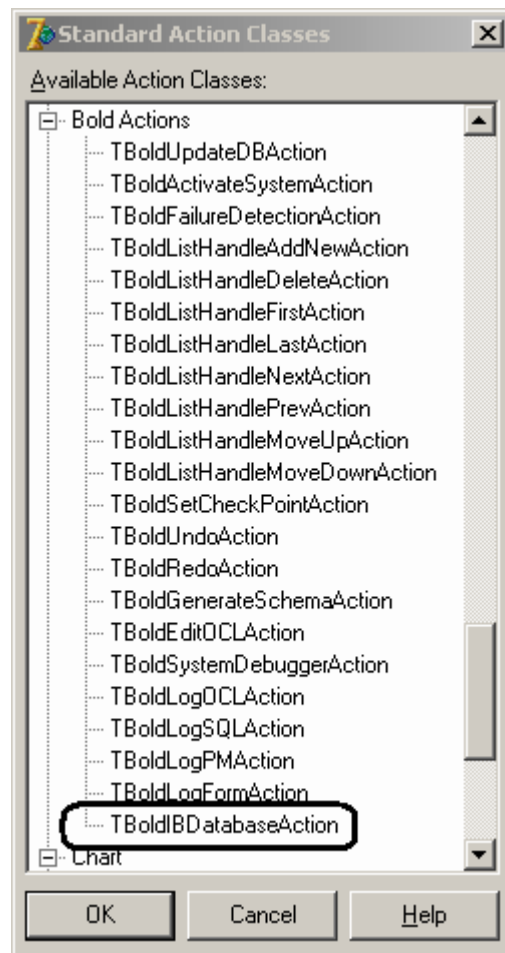


Fig. 10.17. Selecting action from the list

This action is intended for generating the Interbase DBMS scheme. We can set parameters of this action in the object inspector, having specified properties of the generated database (see Fig. 10.18).

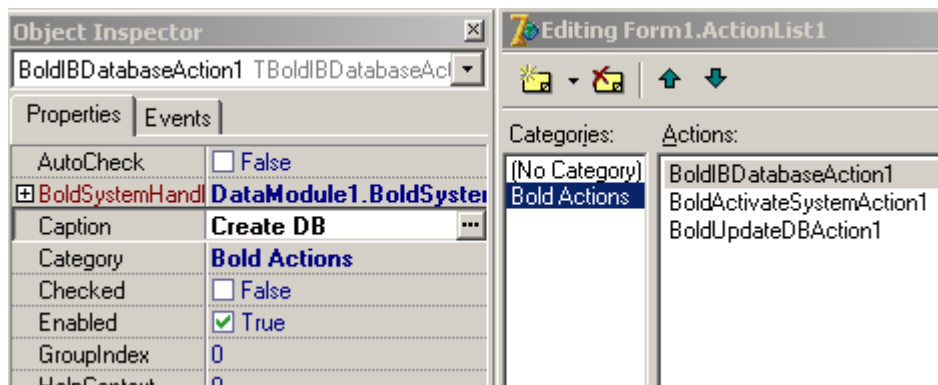


Fig. 10.18. Adjustment of the action properties

Besides we shall add two more actions - `TBoldActivateSystemAction` (object space activation) and `TBoldUpdateDBAction` (database updating). We shall place three buttons on the form (see Fig. 10.19),



Fig. 10.19. Buttons for actions initialization

each of which we shall connect with one of the above-stated actions (see Fig. 10.20), and then we shall start the application on execution.

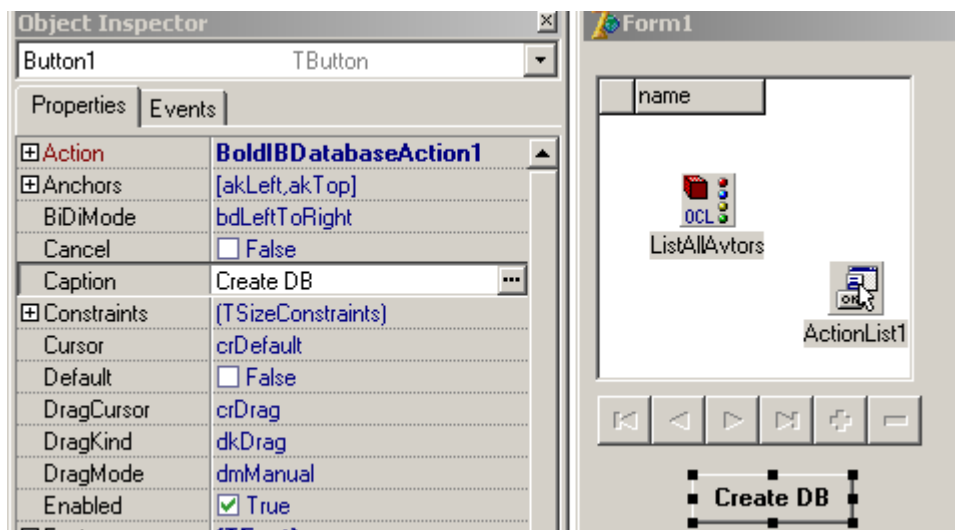


Fig. 10.20. Binding of the button to the action

We shall make sure that when pressing “Create DB” button the new database is generated, when pressing “Activate” button the object space is activated (the information from a DB is mapped and the capability of its editing is provided), and by pressing “Update DB” button the information edited or added by the user is memorized in the database.

We shall note that at BoldActions use there is no necessity to create an empty database beforehand, as we did it in the beginning of the chapter. The only thing we shall do is to specify a name of a file (even if it does not exist at all) of the database in "DatabaseName" property of IBDatabase component.

NOTE

Naturally, if the database file does not exist physically, then it is senselessly to make operations on checking the connection with this DB as it was made earlier.

If at that the way to the file is not set, the database will be generated in the current catalogue. Such capabilities mean that at database generation Bold interoperate directly with InterBase program interface (API) for creating DB file, i.e. “passing over the server”.

NOTE

All demonstration applications - the examples using connection to the DBMS, provided with Bold for Delphi product, are based on the considered capabilities of BoldActions components.

SQL in MDA-applications

SQL Implementation Ideology

Concept of MDA-architecture in general, and its realization in the considered Bold for Delphi product, assumes that all actions with objects should be realized on business layer, at least in the view of ideology. Such advanced capabilities as OCL-queries, navigation on models, "clever" components of GUI, etc., provide the developer with powerful and soft tools for objects control both at the application design stage, and directly at the stage of its execution. Thus from the point of view of an object business layer the persistence layer is auxiliary, and it is located "lower". Therefore, theoretically, the persistence layer is intended only for providing storage of these data between sessions of the MDA-application running.

Such attractive scheme exists in theory, however in practice we have another situation. When creating the "big" and complex applications in practice it is impossible to work without using the tools traditionally related not to business layer, but to persistence layer (DBMS). There are several reasons for such situation, but, evidently, the main reason is **use of two essentially differing models: object-oriented model of the business layer elements and DBMS data relational structure**. We shall consider this moment more in detail. The relational model of the data was developed long ago, and during several decades, which have passed from the moment of the first relational DBMS creation, their toolkit has been perfected and significantly advanced and optimized. Thousands of experts in many companies developed means for increasing SQL-queries productivity for years, and now there are no instruments equal to this tool (SQL) by efficiency of access to the relational data. For optimization the data access relational DBMS create special elements of structure, such as indexes and index tables. At that MDA-application "knows nothing" about these entities, and "not able" to use them.

On the other hand, at least for today, methods of access to the object-oriented data are not worked through and optimized carefully. At present time there are no formalized standards of constructing object-oriented databases, and rules and methods of access to objects and attributes. Therefore in practice situations occur, when externally "object-oriented" DB is based on a relational core, using its capabilities for effective processing the information.

So it is quite obvious that when involving relational DBMS as a persistence layer, MDA-applications "are compelled" to use such effective and debugged instrument, as SQL. Otherwise productivity would fall sharply at working with the data, especially with data bulk. On the other hand, if we consider client-server architecture of databases applications, we shall find out one more reason for use of SQL language. Really, if we deal with the client MDA-application working with a server database of large volume, in practice there will be a following problem: how and where can we realize selection from the database. If we do it in the MDA-client, it will turn out that it is necessary to receive at first on the client side a database copy, and only after this to process these data with the help of OCL-queries. Such requirement occurs in view of the fact that **the object space is always located in memory**. First the described scheme of functioning obviously contradicts the client-server approach, and, second, it is absolutely senseless from the point of view of required resources of a client workstation. The following conclusion can be made: it is expedient and practically necessary to use in client-server architecture capabilities of SQL-

server for data processing, and, hence, the MDA-client should have a possibility for forming necessary SQL-queries.

NOTE

It is necessary to emphasize the fact that the persistence layer in Bold for Delphi ideology includes not only the data or databases, but also tools for these data management, i.e. actually SQL-servers (DBMS).

However it is necessary to understand exactly that after getting a result of such SQL-query, Bold for Delphi environment provides "integration" of the received data into object space (for this purpose there are special components and mechanisms considered below in this Chapter). And after such integration the data received "directly" can be used by the application in the same way as all other elements of object space.

Let's note that in many cases (if the volume of the data in a DB is not large, or in case of using local DB), when creating MDA-applications of databases it is possible to do without using SQL. In each concrete case the developer himself can define necessity of using such "low-level" tool as SQL (in view of business layer object architecture). Below in this section we shall consider the basic tools and capabilities of Bold for Delphi environment for work with SQL.

Using BoldSQLHandle

In Chapter 7, at object space handles review, we have not examined BoldSQLHandle component, having postponed its consideration up to the given Chapter. In this section we shall study capabilities of this component. BoldSQLHandle component is located on "BoldHandles" bookmark of Delphi components palette. Purpose of the given component is the direct reference to the DBMS for fast data selection by means of SQL-queries. For illustrating use of the given component we shall create the simple project on the basis of the example considered above, thus in UML-model we shall leave only one "Author" class. Also for convenience we shall use BoldActions components considered above and the buttons linked to them. We shall add on the form BoldSQLHandle component, which properties we shall adjust as follows (see Fig. 10.22):

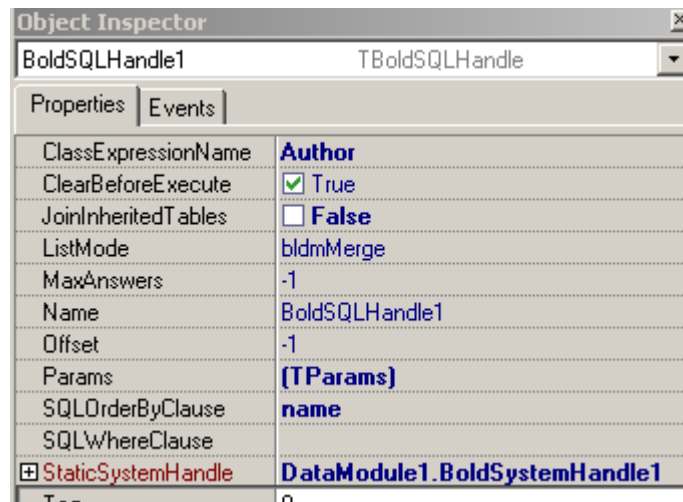


Fig. 10.22. Adjustment of BoldSQLHandle component

- We shall set "Author" class value to "ClassExpressionName" property (it is possible to specify this property manually or by selecting the necessary class from the list of accessible model classes after double click on the given property). The given property defines the element of model, which table-analogue is a source of information for SQL-query;
- We shall set "name" value to "SQLOrderByClause" property. The given property determines «Order By» condition for SQL-query. In this case we are going to order authors by name;
- We shall set BoldSystemHandle1 value to «StaticSystemHandle» property. The given property "connects" SQL handle to the object space used by default. If this property is not set, attempt to make SQL-query active will lead to the program exception.

To compare behaviour of usual OS list handle and SQL handle, we shall place on the form the second BoldGrid component, and the second list handle. It is necessary to note that BoldSQLHandle component is not capable to return data sets, for this reason results returned to it cannot be mapped directly, for example, in BoldGrid, i.e. the component-"adapter" is necessary; the list handles or the cursor handles can be used as such adapter (see Chapter 7). For this purpose we use the second list handle, and we shall set SQL-handle as its root handle ("RootHandle" property). The second BoldGrid we shall connect with the added second list handle. After creation of the columns by default this grid will display the same columns, as the first one (see Fig. 10.23).

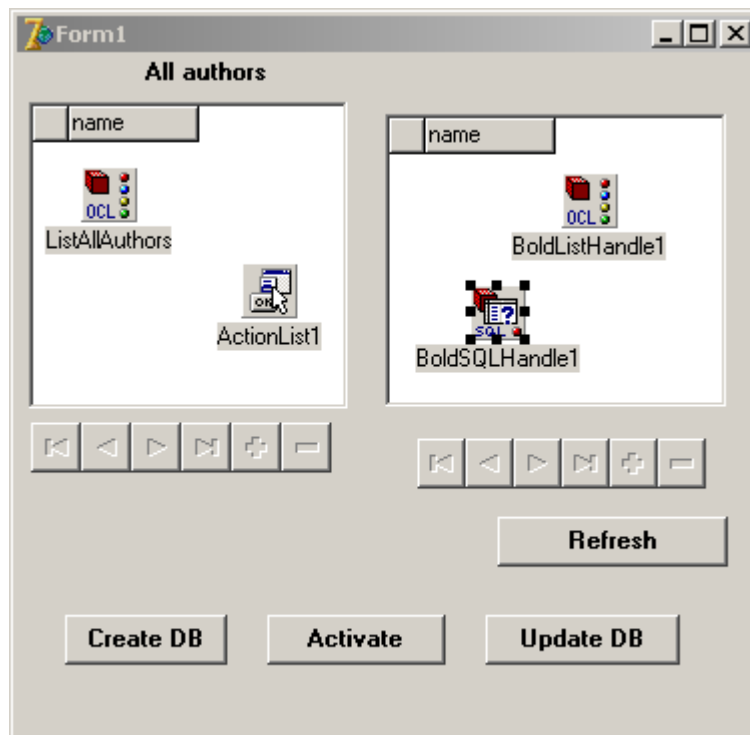


Fig. 10.23.

Thus, first BoldGrid will display the data by standard for Bold way (i.e. by means of receiving data from the OS), and the second grid connected through «list handle - SQL handle» handles chain will obtain the data directly from the persistence layer (DBMS). For activating SQL-query of BoldSQLHandle component it is necessary to call its "ExecuteSQL" method. We shall connect this call to the usual button with "Refresh" name, and enter a simple string of a code into its pressing event handler.

```
BoldSQLHandle1.ExecuteSQL;
```

Let's start the application, and, after DB activation we shall press "Refresh" button. We shall see that in the right grid the same authors, but ordered by names, were displayed .

Let's add with help of Bold-navigator new author in the left grid and we shall press "Refresh" button again. At that nothing occurs, i.e. contents of the right grid will not change.

It can be explained by simple circumstance: the right grid displays the data directly from a DB, and after addition of the new author into the OS we have not made operation of database updating. After pressing "Update DB" button and restarting SQL- query we can make sure that in the right grid the data were updated.

From the considered simple example the difference of "nominal" functioning of Bold application (work at object space layer) and direct "low-level" work with a persistence layer is obvious. At interoperability with OS all changes occurring in it are automatically

“traced” by the application GUI. At direct work with persistence layer the developer should trace such changes. It is necessary to emphasize that a considered SQL-handle component, as it is clear from its properties structure, is intended for realization of the data fetching from the single table, which in DB "maps" the class specified by "ClassExpression" property. We shall consider other properties of the given component. "ClearBeforeExecute" logic property sets necessity of preliminary clearing the results of query before its activation. Rules of results integration of sequential SQL-queries, when setting the considered property in "False" value, are determined by another property of component – "ListMode". For example, if property "ListMode" is set in "Allow" value, and “ClearBefore Execute” property - in "False" value, then after executing two consecutive SQL-queries results of the second query will be added to results of the first one.

If "ListMode" property is set in "Merge", results of sequential queries will be integrated. "MaxAnswers" property allows limiting quantity of recordings returned by query (by default this quantity is not limited, and value of property is equal to 1). This property is convenient to use jointly with "Offset" property, which sets quantity of gated recordings, which will not be returned on the component output. "Params" property allows forming parametrical SQL-query, and its use is similar to usual VCL-DB TQuery component. "JoinInheritedTables" property determines necessity of inclusion of the inherited tables into SQL-query. In a case when the SQL-query uses only own class attributes, this parameter should be left in value by default ("False"). And finally, last property - "SQLWhereClause" allows to add into the SQL-query an additional fetching condition corresponding to "WHERE" SQL-operator. We shall note, that BoldSQLHandle returns the data, which can be edited. It is easy to make sure in it, having added on the form BoldNavigator component connected with SQL handle through the list handle (it is similar to the second BoldGrid) and having started the application. After SQL-query activation the data in the right grid can be edited (to change, delete, and after updating a DB the made changes will be saved), however we cannot add the new author (SELECT query with conditions). The considered component can be applied if necessary to provide maximum fast data fetching, it is especially effective when working in "client - server" architecture.

OCL2SQL Mechanism

In structure of properties of the object space components-handles considered in Chapter 7 - BoldListHandle (the list handle) and BoldExpressionHandle (the OCL-expression handle) - there is the property named "EvaluateInPS" - Evaluate In Persistence Layer. By default the given property is set in "false", thus providing "evaluation", i.e. obtaining the result of the OCL-expressions set in "Expression" property of the given handles, at the business layer. When setting the given property in "true" the mode of direct data processing in the DBMS, with use of so-called "OCL2SQL" mechanism is set. This mechanism of object-relational mapping provides translation of OCL-expressions into operators of SQL-queries. As it was already spoken, use of SQL-queries is of great importance, and it can be applied in many situations. For example, let's imagine the application working with large enterprise staff database. We shall assume that it is necessary for us to select employees, whose age does not exceed 30 years. Then we can generate the following OCL-expression for solving the task

```
Employee.allInstances->select(age<=30)
```

But can we consider the task to be solved? In theory only. And in practice, we shall remember that OCL-expressions evaluation is realized by default at business layer, i.e. in object space, and, hence, in the workstation memory. And in turn, it means that in the considered situation at first all employees of firm should "be loaded" in memory, and only after that necessary fetching will be carried out. Naturally for large data volumes it is illegal, both by speed (network access), by the network traffic, and by required resources (memory). It is much more logical in such situation to make the remote SQL-server of the DBMS the responsible for data fetching operations. However, such transformation-translation has a number of constraints, which are useful for the developer. Sometimes such constraints can cause even the necessity of changing UML-model; anyway, it is expedient to take into account presence of such constraints when creating new model. The reason of the constraints of OCL ► SQL mapping is simple and was discussed earlier, namely, it is essential distinction of object and relational concepts of data presentation. OCL language structures, which are supported by the mechanism of object-relational mapping are listed below and they **can be translated** into SQL-queries:

- ❑ All capabilities of OCL-navigation on UML-model for access to roles and attributes, **except for derived attributes and associations**;
- ❑ Operations for work with collections – select, reject, allInstances, size, orderBy, minValue, maxValue, average, sum, exists, forall, notEmpty, isEmpty, union;
- ❑ Logic operators – =, <, >, <=, >=, <>, and, or, not, xor, sqlLike, sqlCaseInsensitiveLike;
- ❑ Arithmetic operators – +, *, /, -, div, mod;
- ❑ Operations for work with types – oclIsKindOf, oclIsTypeOf, oclAsType;
- ❑ IsNULL operator;

In the following list OCL2SQL constraints are presented, i.e. OCL expressions, which **cannot be translated** into SQL-queries:

- Operations of types transformation and receiving the model metadata – TypeName, attributes, associationEnds, superTypes, allSuperTypes, allSubClasses, oclType;
- Operations of simple types transformations – subString, pad, postPad, formatNumeric, formatDateTime, strToDate, strToTime, strToDateTime;
- ❑ Operations for supporting "history" of the OS – atTime, allInstancesAtTime, existing;
- ❑ Operations with collections – count, includesAll, difference, including, excluding, symmetricDifference, asSequence, asBag, asSet, append, prepend, subSequence, at, first, last, orderDescending, sumTime;
- ❑ Other structures – length, min, max, asString, allLoadedObjects, regexpMatch, inDateRange, inTimeRange, constraints, collect, if, concat

In spite of many constraints, OCL2SQL mechanism allows in some cases to increase cardinally productivity of program system functioning, and, certainly, the developer should apply it actively. This mechanism functions completely in an automatic mode, and it is

imperceptible for the developer. Therefore OCL2SQL use does not cause difficulties, it is enough to set value of the above-described "EvaluateInPS" property of used handle in "True". Bold for Delphi environment will take care of all further system actions with regard to SQL-queries formation.

Optimization of Work in Client-server Architecture

Besides using the components and mechanisms described above, for system functioning optimization in the client-server configuration, additionally, it is expedient to take into account the following recommendations.

3. To avoid OCL-expressions of "Author.allInstances" type. Execution of the given OCL-query will lead to loading of all objects of the given type into the OS, with all following consequences (resources, productivity, traffic). In this case OCL2SQL mechanism "will not help", because the above-stated OCL-expression is mapped in SQL-query of "SELECT * FROM Author" kind with the same consequences. As well as at traditional development of client-server applications, it is necessary to select from DB only those data, which we need at the present moment.
4. For increasing speed of objects set loading it is expedient to use EnsureObjects method of TBoldObjectList class. At that all objects will be loaded from DB for one operating session. Otherwise Bold loads each object during separate sessions.
5. To use "intellectual" capabilities of TBoldGrid component. This component controls loading from a persistence layer so that the quantity of objects corresponded to quantity of visible grid strings. At vertical scrolling a grid loads the next portion of the data from DB.
6. To minimize quantity of OS active handles at the moment of starting the application, and to connect them as required.
7. To use BoldUnloaderHandle (see Chapter 7) for automatic clearing workstation memory.

Bold and "Thin Databases"

Bold for Delphi is powerful tool for fast development of complex applications of databases and information systems. However, you should not think that this product is useful only for decision of large and "serious" tasks of a corporate level. Using Bold for creating small local applications is also justified due to cardinal productivity increase and automation, for example, regarding such tasks as creating forms for data input. Using Bold for Delphi along with the so-called "Thin Databases" is very attractive. In brief we shall explain that "Thin Databases" differ from other local DB by the following: the DBMS core at the project compilation is embedded directly into the application executed file, therefore nothing is required for functioning such applications, neither BDE, nor ADO, nor DLL additional libraries, etc. Thin DB applications can be easily started without any installation, and from any carrier (Flash, CD-ROM, etc.), even without copying to hard disk. Probably, Bold for Delphi developers taking into account such capabilities, have included into DBMS adapters structure the TBoldDataBaseAdapterDBISAM component, which provides interoperability

with DBISAM Thin DB [4]. When using DBISAM together with Bold, complex applications used in small companies can be created in several days. Besides such small "thin" applications can be used for improving and modelling the future "complex" applications (for example to debug design and functionalities of the GUI), and also for creation of working models and delivery presentation variants, with a view of advertising, etc.

NOTE

Recently the free-distributed product similar by functionality has appeared - it is a clone of FireBird DBMS, which name is FireBird Embedded. It also can be related to Thin DB, with all advantages considered above. The described version of Bold for Delphi product is completely compatible with this DBMS. For use FireBird Embedded it is enough to be connected to it by means of IBExpress standard components (see the beginning of the given Chapter). At that, it is possible to involve also BoldActions components considered in this Chapter.

Work with Several DBMS

In practice of applications development situations quite often occur, when it is necessary to port application created earlier for work with another DBMS. At that, naturally, it is desirable not to lose information already written in the database. We shall consider a simple example and show how to solve this problem using Bold for Delphi capabilities. As a basis we shall take the examples of applications functioning with Interbase DBMS, which were examined earlier in this chapter. Let's assume for simplicity that there is a necessity to translate the entire application into Microsoft Access DBMS. What actions should the developer make in this case in order not to lose the data? First, it is necessary to construct "an instance" of components set for work with another DBMS. This instance will be temporary for providing holding the capability of work from an old DB with the purpose of import from it the data entered before. We shall add into the data module one more system handle, the DBMS handle and ADO adapter, having connected it to ADOConnection component (see Fig. 10.28). We shall create empty MS Access database, and adjust corresponding properties of its connection. Further, at the development stage in Delphi environment we shall temporarily connect BoldPersistenceHandleDB2 new persistence layer handle to BoldModel component, and generate the scheme of MS Access database. Thus, our application has got a fundamental capability to work with two DBMS; it is easy to make sure in this fact, for example, having added on the application form two grids and two list handles, at that each handle should use different system handles – bsh1 and bsh2 – as a root handle (Fig. 10.28). Now it is necessary to provide the information transfer. It can be done in several ways; we shall choose the most evident from them (see Listing 10.3).

Listing 10.3. Information transfer from one DB into another

```
procedure TForm1.Button6Click(Sender: TObject);
var i,j :word;
    ob1,ob2:TBoldObject;
begin
    for i:=0 to dm.bs1.System.ClassByExpressionName['Author'].Count-1
    do begin
```

```

ob1:=dm.bs1.System.ClassByExpressionName['Author'].BoldObjects[i];
ob2:=TBoldObject.Create(dm.bs2.System);
for j:=0 to ob1.BoldMemberCount-1 do
ob2.BoldMembers[j].Assign(ob1.BoldMembers[j]);
dm.bs2.System.ClassByExpressionName['Author'].Add(ob2);
end;
dm.bs2.UpdateDatabase;
end;

```

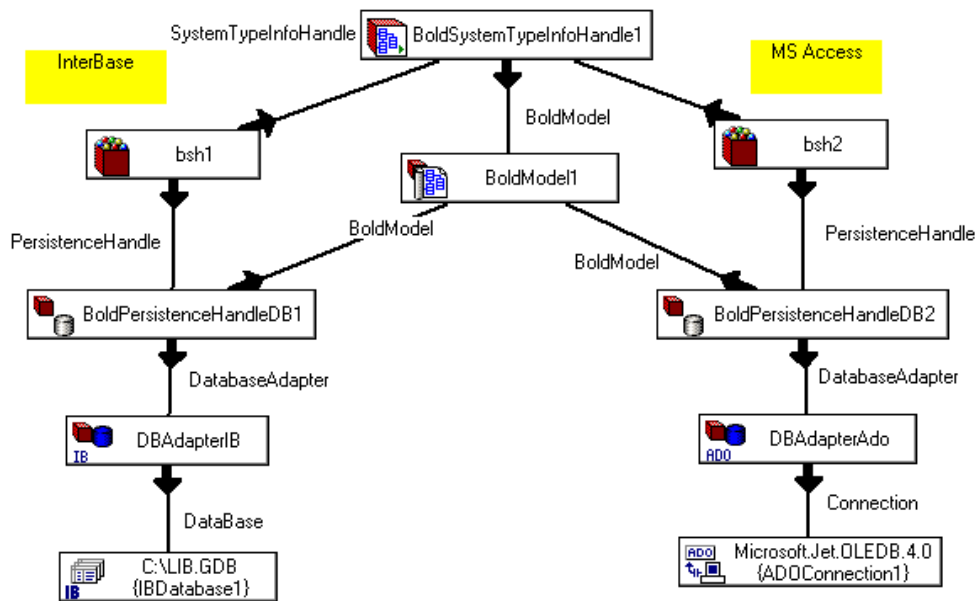


Fig. 10.28. Application components diagram for the case of two DBMS

The sense of program operations is rather simple and evident, step-by-step object copying between different OS (old and new), and addition of object into the list corresponding to the new database is used. After that it is enough to call UpdateDatabase method and all the information will be fixed at the persistence layer of the new DBMS. Then it is possible to remove the components responsible for "old" Interbase DB from the application, and to continue the work with MS Access. Based on this approach it is easy enough to make simple procedures, automating the data transfer for more complex applications too. Thus, the main advantage of Bold for Delphi program system at realization of such operations is that work is carried out at the business-layer (a layer of object space); and when creating procedures of the information transfer (see Listing 10.3) we can do without knowledge of concrete structure of a database as Bold takes care about the task of OS structure transformation into relational structure of the DB information.

Using XML-documents

Data Saving in XML-files

In this book we more than once used `TBoldPersistenceHandleFileXML` component (see Chapter 3). This XML-files component-handle provides holding the object space state in usual text file of XML format. And this occurs without constraints regarding the kind of information, i.e. at that we can save not only the text data, but also, for example, photos, Word documents or Adobe Acrobat PDF-documents, etc.

NOTE

Naturally, for the listed types of documents it is necessary to generate in UML-model special attributes of BLOB type or its subtypes.

The way of data storage in XML file has the advantages and shortcomings. The main advantages are the following:

- ❑ Simplicity of development process. There is no necessity to use DBMS, to adjust connection to the DB, to generate the DB scheme, etc.
- ❑ Simplicity of use. The maximal transferability of the application is provided, there is no necessity of installation and adjustment, start of the application from any carrier on any computer is possible.

The shortcomings of using XML are the following:

- ❑ Low execution efficiency at data bulk;
- ❑ Impossibility of SQL use. Impossibility of OCL2SQL mechanism use;
- ❑ Impossibility of work in client-server configuration.

However available constraints do not prevent extensive use of XML for development of small office or home applications, reference books, presentation applications, etc. In this sense the listed advantages are close enough to the advantages of using “thin databases” considered in the previous section of the given chapter

NOTE

However, it is necessary to have in view that “thin databases” support SQL, and, owing to this reason, have significant advantages due to processing speed.

Practical Example of XML Use

We shall illustrate practical use of XML files by the simple, but a little bit unexpected example. Namely, we shall pose the following task: it is necessary to create program library (module); its set of procedures should provide retention of state and size of the application forms between sessions. We shall show, how can we solve the posed task; at that we shall

present only the solution scheme, illustrating the basic approaches, however, it is quite sufficient for the independent complete solution.

In this case the UML-model will consist of a single class, which attributes describe the form state and sizes (see Fig. 10.28).

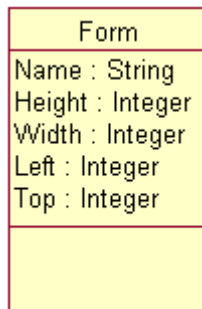


Fig. 10.29. Class providing form state saving

We shall name the class as "Form", and attributes – by analogy with modeling properties of the Delphi-form, i.e. "Height", "Width", "Left", "Top". Besides for binding to the concrete form of the application, we shall add "Name" attribute, i.e. a name of the form. For debugging our library we shall create the new simple Bold-project containing the empty form and the data module. We shall add on the data module necessary components (system handle, model, types handle and XML-files handle, and then we shall adjust them in the same way that we did when creating the simple application (see Chapter 3). We import the created model containing a single class from Rational Rose into Bold for Delphi.

NOTE

In this case it is easier to create such model in built-in Bold editor, therefore this step is presented only for completeness of the scheme of the task solution

Now we shall start creation of procedures. It's enough to have only two procedures: one procedure should provide saving the form properties, and another – loading of the saved parameters from XML file. The program code of the first procedure is presented below (see Listing 10.4).

Listing 10.4. Form State Saving Procedure

```
Procedure SaveFormState (F:TForm);
var O:TBoldObject;
begin
```



```

O:=ObjectLocate('Form','name',f.name);
if O<>nil
then EditObject(O,['name','left','top','height','width'],
               [f.name,f.left,f.top,f.height,f.width])
else NewObject('Form',['name','left','top','height','width'],
               [f.Name,f.left,f.top,f.height,f.width]);
end;

```

NOTE

Procedures presented in this section use a number of auxiliary universal functions and procedures for work with object space; their description and sources are given in this book in the [Appendix](#).

Sequence of actions made in the given procedure is following.

- ❑ It is checked, whether there is a record about the given form. For this purpose ObjectLocate function is called; it returns required object if it is found, or NIL value if such object is absent;
- ❑ If the object-form is found, its attributes are edited; the current parameters of the form are set as attributes. EditObject procedure is applied for this purpose;
- ❑ If record about the given form is entered for the first time, "Form" new class object is created; and the current parameters of the form are set to its attributes.

Below (see Listing 10.5) a program code of the second procedure providing loading the form state from the XML-document is presented.

Listing 10.5. Form State Restore Procedure

```

Procedure LoadFormState (F:TForm);
var O:TBoIdObject;
begin
  O:=ObjectLocate('Form','name',f.name);
  if O<>nil then with F do
  begin
    left:=Attr(O,'left');
    top:=Attr(O,'top');
    height:=Attr(O,'height');
    width:=Attr(O,'width');
  end;
end;

```

Sequence of actions made in the given procedure is following.

- ❑ It is checked, whether there is a record about the given form. For this purpose ObjectLocate function is called; it returns required object if it is found, or NIL value if such object is absent;
- ❑ If the object-form is found, attributes values of the found object are set to the current parameters of the form state. Attr function, returning attribute value of the object on its name, is applied for this purpose.

For using the created procedures in other applications, it is necessary to choose them together with other involved components into the separate program module (Unit) and to provide their access by carrying out declarations of these procedures in the interface part of the received module. Now it is enough to connect the created module at the development stage to any application, and to call created procedures in events handlers (for example, events of creation and the form closing) for those forms, which status is required to be remembered between the application sessions.

NOTE

At desire, it is possible to extend essentially structure of the saved parameters of the form, having added, for example, attributes of color, fonts, etc., and, theoretically, saving a status of all child components of the form. In this connection the considered example is not only abstract. Now active development is conducted in the field of using XAML (XML Application Language), which is planned for using as universal tool for description of the applications GUI in some future operational systems.

As we can see from the presented example, the field of implementation of XML-documents as the data "storehouse" is rather wide.

Summary

In the given chapter work with Persistence Layer is described. The wide spectrum of Bold for Delphi environment capabilities regarding use of relational DBMS is shown by concrete examples. "Transparency" of the databases structure generated by Bold environment is shown. Importance of SQL is shown, when developing client-server applications, and the basic components for work with this language are described. The second way of data storage, using XML-documents, is described; in some cases it is also the useful tool of the applications creation.

Chapter 11. Using Third-party Visual Components

Visual components for GUI creation forming Bold for Delphi (see Chapter 9) provide the developer with a wide spectrum of capabilities on creating the user interface, providing at that convenient and unique resources regarding automatic synchronization of information graphic representation with object space (OS) state. However, during applications practical development the necessity of using third-party components often occurs, for example, for receiving new functionality of the interface, increasing convenience, for design improvement, or for supporting unified style design of the application interface.

NOTE

By now dozens packages of components for Delphi providing formation of rather expressive, attractive and multipurpose graphic interfaces are developed. Software products of DeveloperExpress, LMD, TMS Software and some other companies can be cited as examples.

At that there is a natural question: how to provide such third-party components functioning in the applications created by means of Bold for Delphi? Really, as against visual Bold-components, third-party components “know nothing” either about object space, or about OCL. These components have no property such as "Expression", and cannot receive the information from OS handles. The posed problems directly concern using the standard VCL-components, which are included in Delphi distribution. For example, using Bold for Delphi capabilities the developer can quickly create the working application of databases; but how can he provide after that formation of beautiful and evident pie charts by means of DBChart standard component? In this chapter we shall answer the posed questions. At once we should say that there are no any fundamental difficulties at use of third-party components packages in MDA-applications. Bold for Delphi developers, naturally, have provided variants of using this technology for functioning in such "mixed" configuration, when for GUI formation it is necessary also to involve components of third-party companies.

External Components Controlling Means

TBoldPropertiesController component intended for controlling properties of external components is specially included into Bold for Delphi components structure. It is located on <BoldControls> bookmark of Delphi components palette. A principle of the given component operation is the following: it allows to connect a set of external components to itself, (and it's not of necessity that these components to be located on the same form!), to

define concrete property for each connected component, and after that, during the application run the given component automatically forms values of the set properties of the specified external components, giving to them an OCL-expression result (this expression is set in "Expression" property of the considered component). We shall illustrate by the example, how does the given component work. For this purpose as a basis we shall use the simple application created in the previous chapter, on main form of which we shall leave only BoldGrid1 with the navigator, and BoldActions components control buttons (see Chapter 10). We shall add Label2 standard component on the form. And then we shall add BoldPropertiesController1 component on the form, and set its properties as follows (see Fig. 11.1):

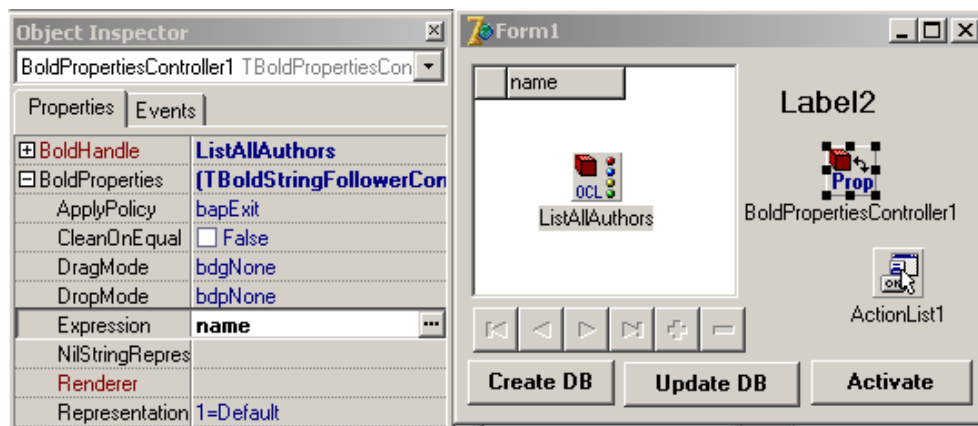


Fig. 11.1. Properties controlling component adjustment

- ❑ We shall set ListAllAuthors authors list handle as a handle (BoldHandle property);
- ❑ We shall set author's "name" as OCL-expression (Expression property).

After that we shall double click on DrivenProperties in the object inspector (or on BoldPropertiesController component), and we shall enter in the editor of properties control elements. This editor is arranged by standard manner. It has buttons for elements addition and removal. We shall add a new control element (see Fig. 11.2).

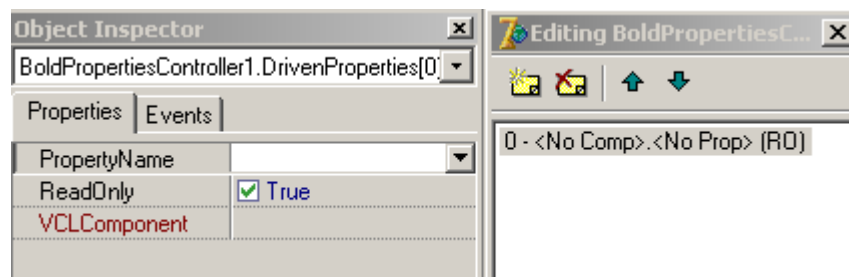


Fig. 11.2. Properties controlling elements editor

Further, we shall pass into the properties editor and open "VCL-component" property (see Fig. 11.3) for binding the external component to the new element. We shall see that in the components list all components of all application forms are represented, that is rather convenient (it does not require manual input). We shall select Label2 component as "the subject of control".

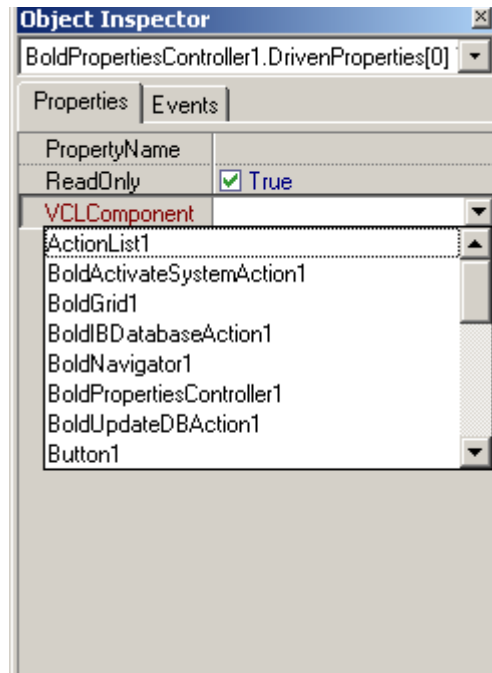


Fig. 11.3. Setting VCL component from drop-down list

Now we shall set concrete property of Label2, which should be controlled by BoldPropertiessController1 component. For this purpose we shall open "PropertyName", and from the available properties list (this list is also formed automatically) of the selected component we shall select "Caption" property (see Fig. 11.4).

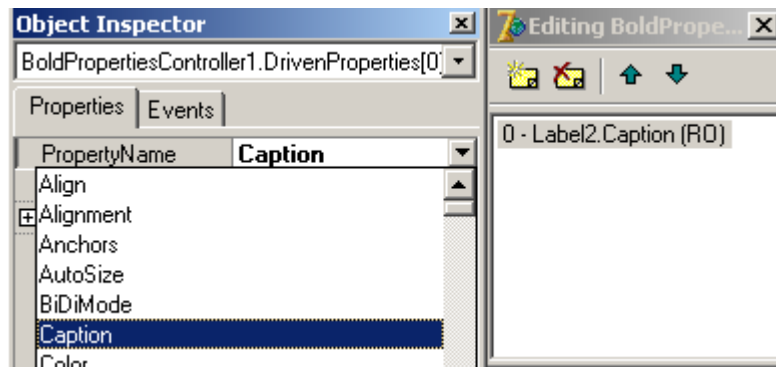


Fig. 11.4. Selecting property for control

After that adjustment of the single controlling element of our component can be considered to be finished (see Fig. 11.5). The essence of the realized operations consists in the following – to provide automatic mapping the current author name by the standard label text.

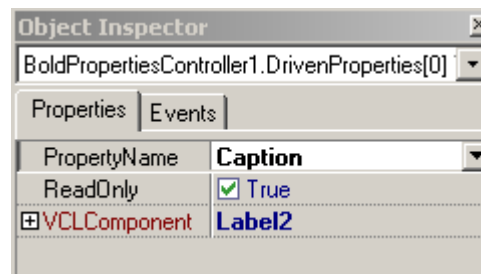


Fig. 11.5. The adjusted controlling element

Let's start the application running (see Fig. 11.6), and we shall make sure that when moving in the list of authors the standard label maps a name of the current author.

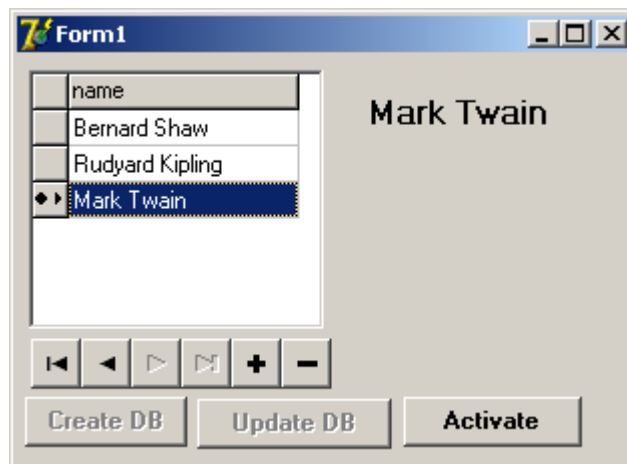


Fig. 11.6. Standard label property control

We have considered the elementary example of using BoldPropertiesController component. Actually, its capabilities are much wider. Using the given component, it is possible to provide easily, for example, automatic synchronization of the selected visual properties of the various components located on different forms of the application. It is important that thus all similar problems can be solved without using program code. Thus, as a matter of fact, the examined component fulfils the functions of the specific intellectual "adapter" providing the information transfer from object space into the environment of VCL usual components.

BoldDataSet– Gateway for Using DB-components

BoldDataSet is one more important component, which provides interoperability with third-party components. This component plays a role of an original gateway to any external visual components intended for operations with databases in Delphi, as BoldDataSet is the successor of well-known TDataSet class. For this reason the given component is capable to replace such components as TTable, TQuery, TAdoTable, etc. BoldDataSet component is located on <BoldMisc> bookmark of Delphi components palette. We shall examine how this component is used in practice. Using one of the previous examples as a basis, we shall modify it a little. We shall remove BoldGrid, BoldNavigator from the form, and instead of it we shall add on the form DataSource component, and BoldDataSet new component. BoldDataSet component has BoldHandle usual property, to which we shall set ListAllAuthors value of the authors list handle. Also we shall set "AutoOpen" property of this component in "True" value for providing its activation by the first request. Other components we shall adjust in the usual way (see Fig. 11.12).

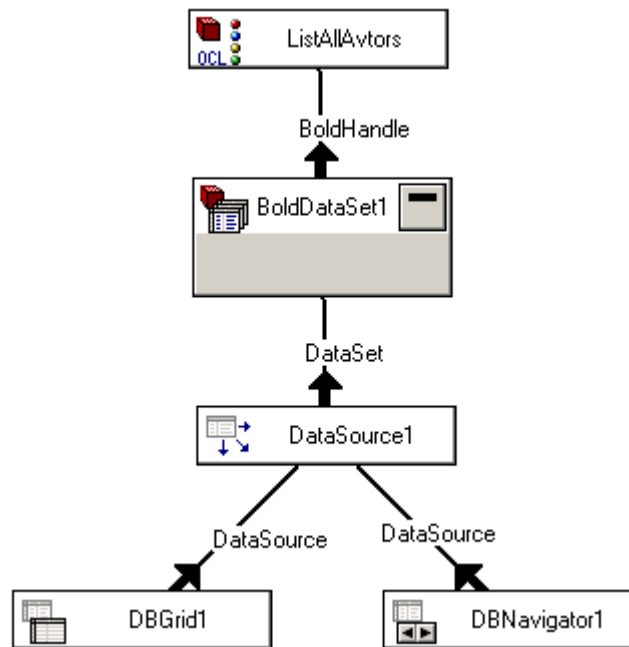


Fig. 11.12. Components diagram

Now we shall set fields of our component. After double click on it the standard editor of fields will open (see Fig. 11.13). We shall add one field - "Author", and set its properties in the object inspector. Each field of BoldDataset component has individual "Expression" property for OCL-expression input. At the present moment we shall enter "name" expression, and it will mean that in the given field the information on the author surname will contain. Besides it is necessary to assign a new field name ("FieldName" property), which we shall name "Author".

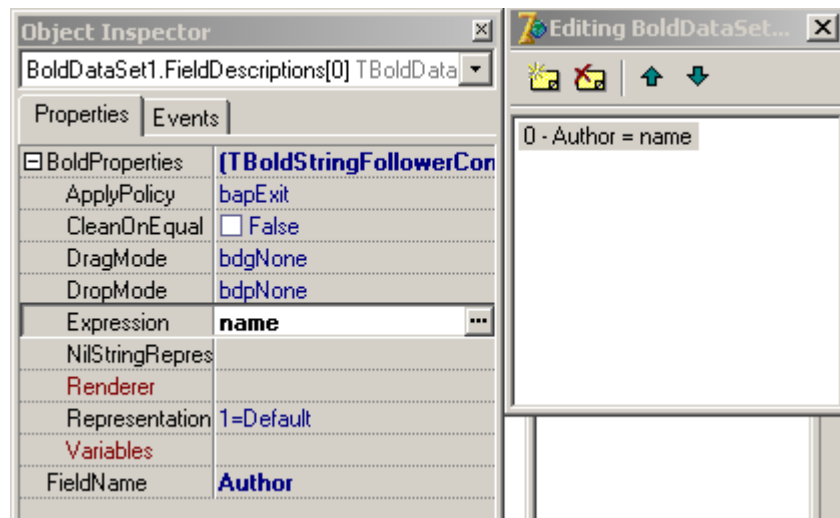


Fig. 11.13. Setting BoldDataSet field properties

After double click on DBGrid1 we shall add one column by the traditional approach, and connect it with "Author" field. We shall start the application, and we shall make sure that it is quite operable (see Fig. 11.14).

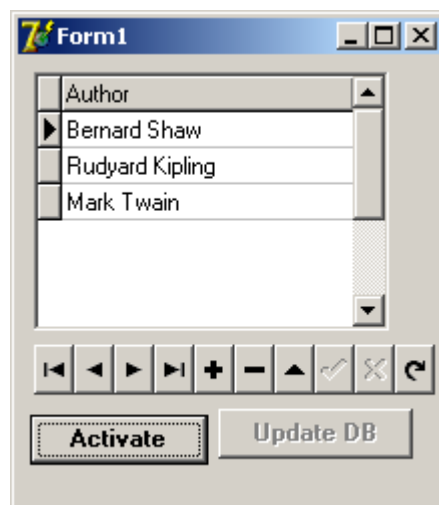


Fig. 11.14. The application with standard components

We can edit, delete and add authors, and after updating the database, the entered changes will be saved. As a matter of fact by this simple example the possibility of full exception of visual Bold-components from the application with keeping functionality is shown. Moreover, at that capabilities of GUI synchronization are also kept. In order to make sure in it, we shall simply add BoldLabel1 on the form and connect it to the same authors list handle – ListAllAuthors (at that having set "name" OCL-expression for its "Expression"

property). After starting the application we shall make sure that the label traces transfers in the authors list (see Fig. 11.15).

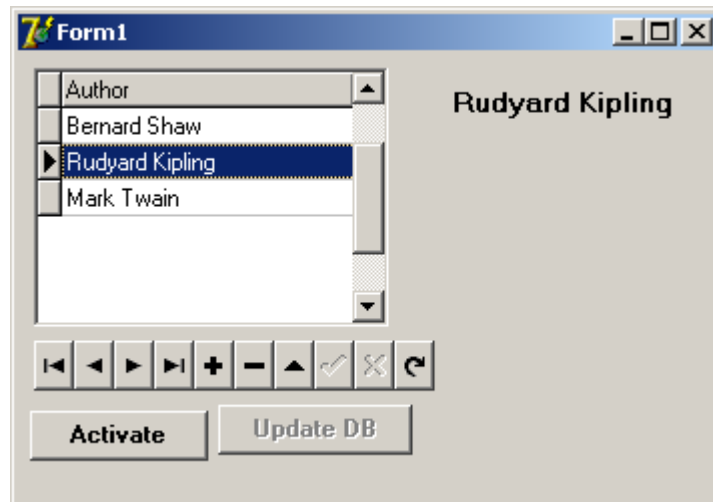


Fig. 11.15. Components synchronization

NOTE

However, some loss of functionality is inevitable when using third-party (and standard) components. The autoforms initiation, Drag-n-Drop support, and some other capabilities will be lost. Nevertheless, they can be realized by program approach too.

Using BoldDataset has also some other advantages. As well as for BoldGrid component considered earlier (see Chapter 9), the effective utilization of the mechanism of navigation on model is the basic and qualitative difference of the given component from standard analogues. It allows, even using third-party visual components such as DBGrid, to provide mapping in one grid practically any necessary information, from any model classes (naturally, these classes should be linked by associations). In our example we map the information on authors, i.e. finally from the DB table with "Author" name. If we used a standard component such as TTable, we would not go beyond such capability. But, as we use BoldDataset Bold-component as a source of the information we can map any information from other (connected) classes, for example, the information on the country. For this purpose we shall simply add a new "Country" field and enter into its "Expression" property (see Fig. 11.16) the following navigating OCL-expression

`country.name`

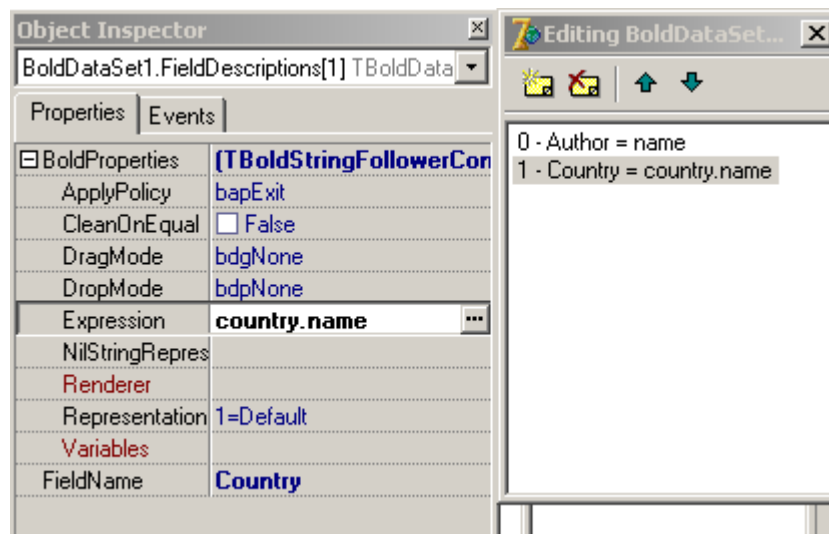


Fig. 11.16. Adding and adjusting the second field

Let's add one more column in DBGrid and connect it with a new "Country" field. Having started the application for running, we shall make sure that the standard grid maps now the data from several tables (see Fig. 11.17).

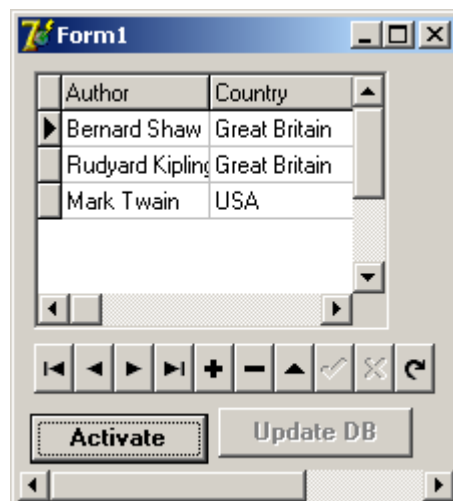


Fig. 11.17. Mapping data from different tables

NOTE

There is nothing essentially new in the fact of mapping the data from several tables in one DBGrid component; the same effect can be achieved, for example, by forming SQL-query and connecting this grid to the corresponding TQuery component. It's quite another matter that in the given example this task is solved much easier and more elegantly.

Thus, it is possible to make the following conclusion: using BoldDataset component allows applying any DB-aware components for the application GUI construction, keeping at that a significant part of advantages of the considered MDA-technology. In practice this component is very convenient for using, for example, jointly with TDBChart visual component for graphic diagrams construction.

ATTENTION

In the examined Bold for Delphi version in BoldDataset component realization developers made an error. It occurs when "AutoOpen" property is set in "True" value and appears in failures when starting the application (loss of the field information). For avoidance of such undesirable effects it is recommended at the development stage to set both "Active" and "AutoOpen" properties in "False" value, and to initialize BoldDataset with the help of the program after starting the application by means of setting "Active" property in "True" value.

Let's note that such functionalities when using the components described above are achieved by means of fundamental division of Bold for Delphi program environment components into three different layers – persistence layer, business layer, and representation layer (GUI). At that each layer solves its own tasks. Replacing Bold GUI layer by third-party components, we "do not touch" a business layer in any way, and it effectively as before operates object space, provides OCL-expressions interpretation, navigation on models, interoperability with the DBMS, and so on. At that by means of the components-"adapters" considered above the business layer also provides transfer of the necessary information for functioning third-party components.

NOTE

By virtue of stated above, it will be a mistake to consider that, for example, DBGrid use results in direct interoperability of the graphic interface with the database table. Actually BoldGrid replacement on DBGrid impacts only "mutual relations" of the graphic interface and business layer. At that all mechanisms of information interchange between Bold business layer and persistence layer (DBMS) "will not feel" such replacement and will continue to function in a usual mode.

Using Subscribing Mechanism and Program Code

The components using the subscribing mechanism embedded in Bold for Delphi can be applied as one more way of realizing interoperability of business layer and third-party visual components. In detail we shall consider this mechanism later, and here we shall show some potentialities of such approach. For this purpose we shall consider BoldPlaceableSubscriber component from <BoldMisc> components palette bookmark. Generally speaking, the given component is intended for realization of some other functions concerning directly the subscribing mechanism, but, nevertheless, here we shall show that it can be successfully applied for other purposes - namely, for synchronization of GUI when using third-party components. We shall pose the following task: to map in TImage standard component a photo of the current author, at that photos should have "jpg" format. For solving this task at first we shall change our model, and enter into "Author" class new attribute named "pic" of TypedBLOB type, in which we shall hold authors photos. This operation can be made directly in the built-in editor of models (see Fig. 11.18).

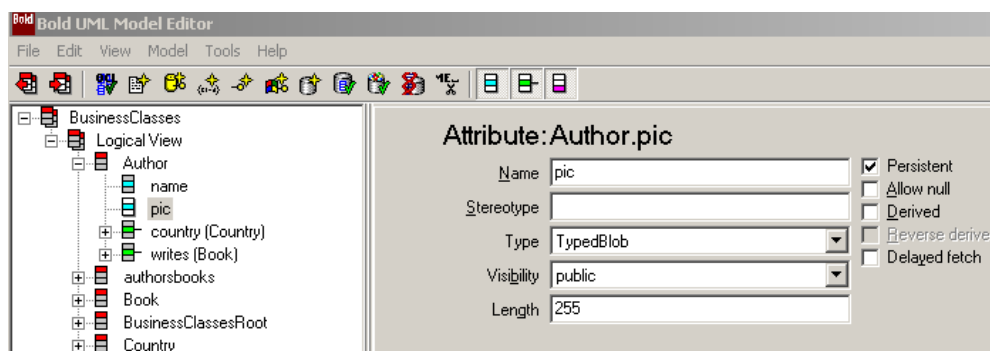


Fig. 11.18. Adding new attribute

Let's add on the application form two components for mapping images: one standard component – Image1, and one Bold-component – BoldImage1 (the second component is useful to us for entering authors photos into the database by the most simple way). We shall add on the form BoldPlaceableSubscriber1 component. Besides we shall add standard component-dialog of opening images files – OpenPictureDialog1 and one button – Button1, having supplied it with a title: “Change photo” (see Fig. 11.19).

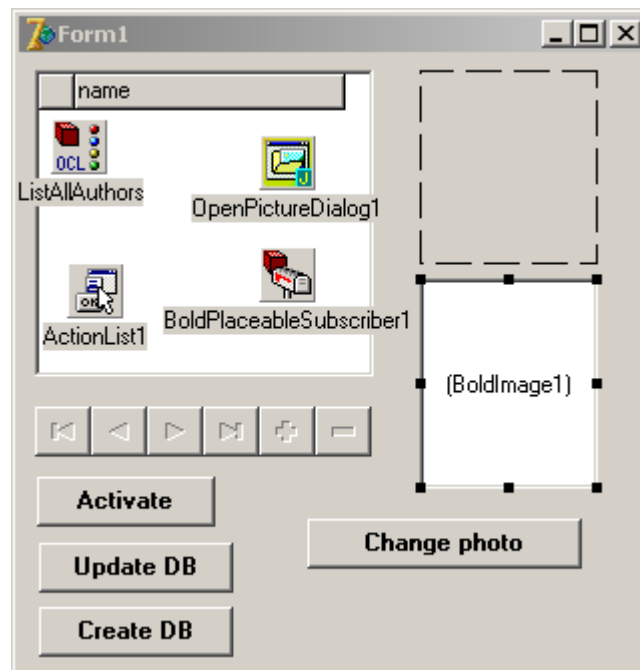


Fig. 11.19. Components lay-out on the form

As we are going to save photos of authors in JPG format, we shall add modules call with "JPEG" and "BoldImageJPEG" names in Uses section of our module (see Chapter 9). We shall connect BoldImage1 component to ListAllAuthors authors list handle, and then we shall specify an OCL-expression for its "Expression" property - "pic". For entering new photos in "pic" attribute we shall use a convenient method of TBoldImage-LoadFromFile class; for this purpose in the event handler of "Change photo" button we shall enter the following code:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute
    then BoldImage1.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Let's start the application run, and load photos of authors from files prepared beforehand, then we shall update the database for saving introduced changes (see Fig. 11.20).



Fig. 11.20. Photos are loaded into DB

Let's connect `BoldPlaceableSubscriber1` component to `ListAllAuthors` authors list handle (`BoldHandle` property). We shall enter the following code string into "OnSubscribeToElement" processing event

```
DrawJPGFromBold(element, 'pic', Image1);
```

By means of this code string the following procedure call is realized

```
DrawJPGFromBold(E1:TBoldElement; attr:string; Im:TImage),
```

which has the parameters:

- ❑ `E1` – parameter of `TBoldElement` type, specifying the object space element, from which it is necessary to load the image;
- ❑ `attr` – the name of an attribute, which contains the image;
- ❑ `Im` – the name of a component of `TImage` type, to which the received image should be appropriated.

The program code of used procedure is presented below (see Listing 11.1) for an illustrating methods of operations with BLOB-attributes. Procedure uses `TBABlob` (Bold BLOB-attribute), `TBoldBlobStream` (Bold-stream providing capability of BLOB-attributes storage) types, and also well-known `TBoldObject` class (see Chapter 6). Main principles of the procedure functioning are quite clear from the listing presented below; there are necessary explanations in comments (see Listing 11.1).

Listing 11.1. JPG-picture loading procedure from Bold-attribute into Image

```
procedure DrawJPGFromBold(E1:TBoldElement; attr:string;Im:TImage);
var bs : TBoldBlobStream;
    sm : TBoldBlobStreamMode;
    bl : TBABlob;
    jpg : TJpegImage;
begin
  if Assigned(E1) then // check, if OS element exists
  try
    // attribute value assignment to bl variable
    bl:=(E1 as TBoldObject).BoldMemberByExpressionName[attr] as TBABlob;
    sm:=bmReadWrite;
    if not (bl.IsNull) then // check that the picture is not empty
    begin
      bs:=bl.CreateBlobStream(sm); // Bold BLOB-stream creation
      jpg:=TJpegImage.Create; // image temporary instance creation
      jpg.LoadFromStream(bs); // image transfer
      im.Picture.Graphic:=jpg; // image assignment to component
      jpg.Free; // temporary instance deletion
    end;
  finally
    bs.Free; // BLOB-stream deletion
  end; // try
end; // proc
```

Let's start the application run, and we shall make sure that the posed task is solved (see Fig. 11.21). Now, when moving in BoldGrid1, Image1 standard component maps a photo of the current author.

The considered example of using BoldPlaceableSubscriber component once more time shows that the application business layer behaviour does not depend on used GUI components. It is enough to be connected to this layer, and the same event synchronization of GUI elements, which is present when using Bold-components, is provided.



Fig. 11.21. Images and grid synchronization

Advantage of using BoldPlaceableSubscriber component in comparison, for example, with BoldPropertiesController component considered earlier (see the beginning of the given Chapter) consists in the following: in BoldPlaceableSubscriber component event handler the developer can write any program code, which will provide necessary flexibility and functionalities. As an illustration we shall replace the handler written earlier with the following code:

```
procedure TForm1.BoldPlaceableSubscriber1SubscribeToElement(
  element: TBoldElement; Subscriber: TBoldSubscriber);
var Ob: TBoldObject;
    st:string;
begin
  st:=(element as TBoldObject).BoldMemberByExpressionName['name'].asString;
  if st<>'Lermontov' then DrawJPGFromBold(element,'pic',Image1)
  else ShowMessage('You have selected the document with restricted access');
end;
```

In the presented code fragment the name of the current author is checked. If the author with "Lermontov" name is selected, then the simple message is displayed (see Fig. 11.22).



Fig. 11.22. Subscription event

By means of such program approach it is possible to generate easily any required logic of the application GUI functioning, and thus flexibility of considered BoldPlaceableSubscriber component is shown.

NOTE

In the given example it is possible to use DBGrid standard component instead of BoldGrid equally well, as it has been shown earlier in this chapter at the description of work with BoldDataset component.

Summary

Bold for Delphi program system has in its arsenal of the means effective and multipurpose components for work with the third-party (external) packages of components intended for GUI generation. For automatic control of external component properties it is expedient to use BoldPropertiesController special component. With the purpose of a possibility of using the external components intended for mapping the information from databases, in Bold for Delphi structure BoldDataset component presents, which is capable to replace any standard or third-party component-descendant of TDataset class. And, at last, for realizing the most flexible program controlling the external components behaviour we can use BoldPlaceableSubscriber component. Use of external components of the GUI does not impact functionality of Bold business layer, at that almost all capabilities of the application control are kept.

Chapter 12. Generating Code

At studying Bold for Delphi environment capabilities, earlier in this book the examples without using code generation of model classes were considered. If we look attentively at Delphi program modules (Units) created in these examples, we will found out that practically they does not contain a program code, except for events or procedures handlers specially created by us. The basic functionality of the application at such variant of development (without generating model classes code), is provided by means of Bold for Delphi built-in program mechanisms. Thus, in the examples considered earlier Bold program system represented itself as "interpreter" of UML-model (see Chapter 1). At that, having the information on UML-model at the stage of the application run, Bold environment "in a hurry" formed necessary links and methods for access to the information we need, provided OCL-navigation and realization of other useful functions. It is necessary to say that in many cases such "interpreter" approach is quite justified, and it is possible to create complicated applications, not using at that generation of model classes. Program modules Delphi received at such approach will be rather compact, easy in maintenance, and the basic logic of work "will not be scattered" over various fragments of a program code. However, all the arsenal of Bold capabilities becomes accessible only when generating classes code. Flexible work with subscribing mechanisms, implementation of operations in model classes, use of reverse-derived attributes - these and other capabilities appear only at Bold code generator involvement. Intuitively it is clear that programming with use of program descriptions of all model elements provides the maximal flexibility when developing applications.

NOTE

It may seem that the code generator involvement for receiving model classes program code brings near the considered Bold for Delphi product and CASE-systems. However, as we shall see further, even when working with model classes code, all "interpreter" capabilities considered before hold in full. It can be explained by the fact that the generated program code represents as a matter of fact program "wrapper" around Bold internal methods and classes, which form the basis of this environment functioning. Therefore the generated code does not replace the "interpreter" functionality realization, but only gives the convenient program interface for the developer.

Further, in this Chapter, we shall consider issues of using code generation at applications development in Bold for Delphi.

Generation Procedure

Let's take the example of the simple application considered in the previous Chapter as the basis, and transform it for a variant of using the program code of model classes. The UML-model of the application consists of three classes and two associations (see Fig. 12.1).

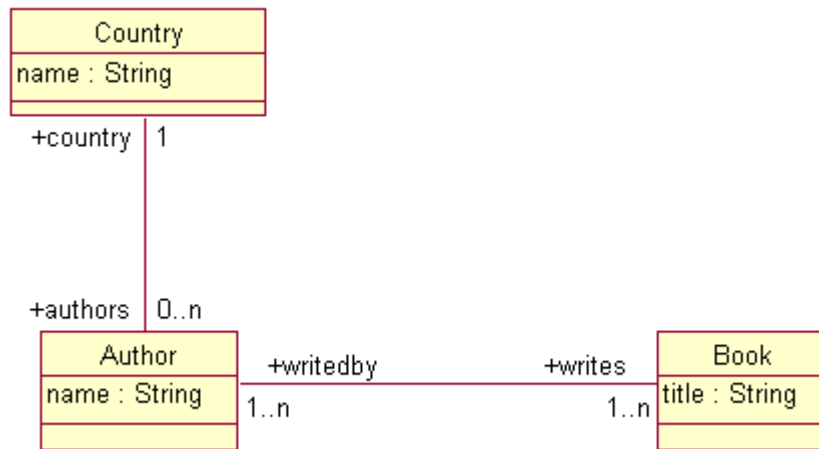


Fig. 12.1. UML-model

For code generation it is enough to call the built-in models editor, and to select in the main menu Tools ► Generate Code items, or to push the icon with sheet and yellow asterisk image in toolbar (see Fig. 12.2).

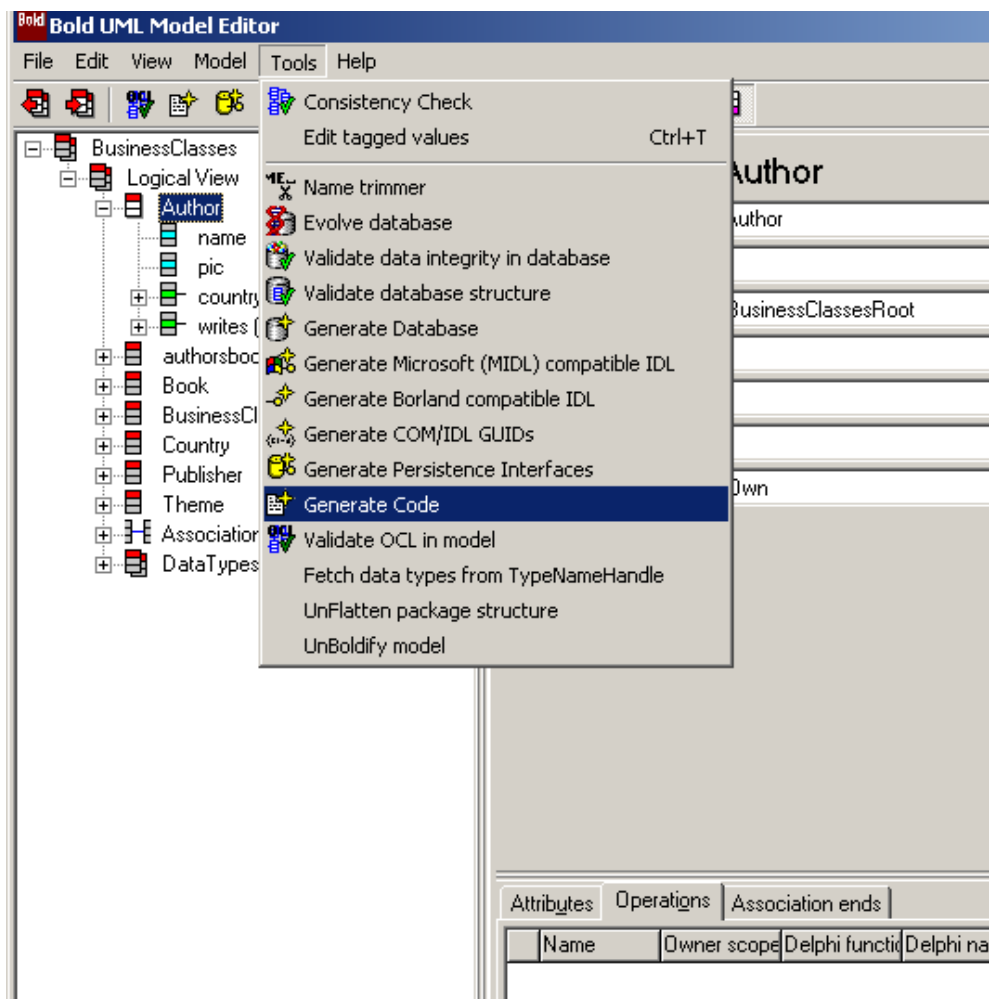


Fig. 12.2. Starting code generation from the built-in editor

Two windows with the proposal to select names of the interface generated files (see Fig. 12.3) and program file will appear consistently. We shall keep the proposed files names without changes.

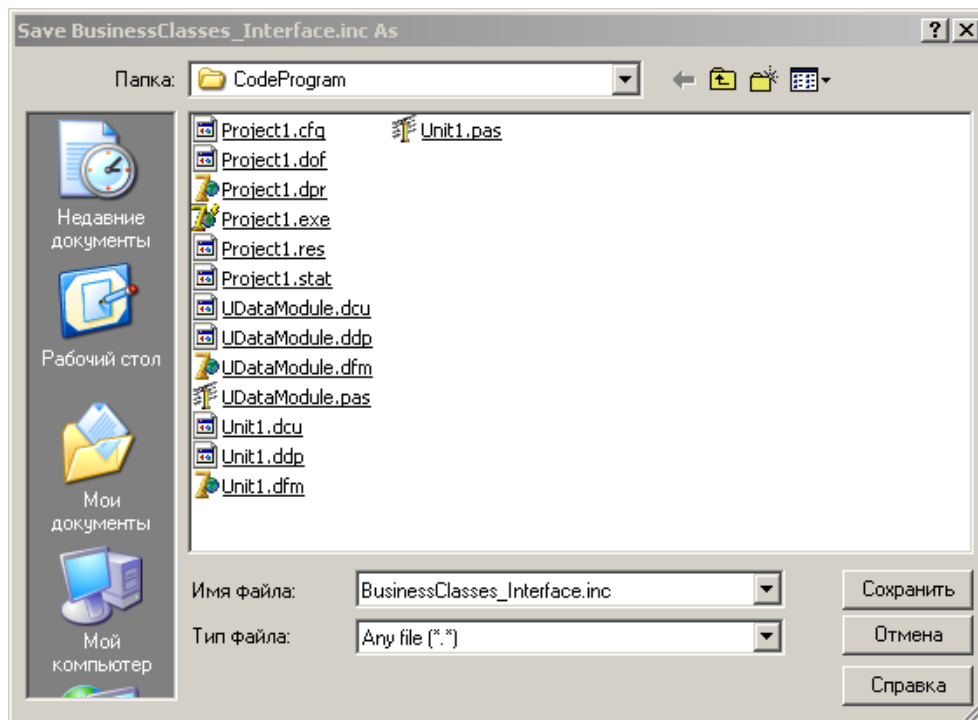


Fig. 12.3. Setting the name of the generated file

Actually, code is generated very quickly, and after finishing this operation the warning will appear in the bottom part of the built-in editor (see Fig. 12.4). This warning notify about necessity to familiarize with the generation protocol; for that you should press the yellow bubble image.

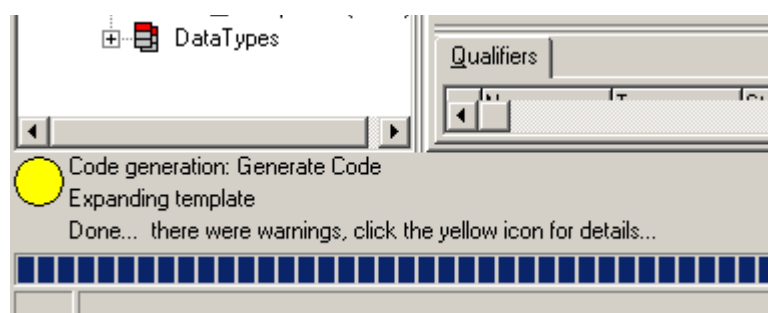


Fig. 12.4. Warning message

As a result of such pressing the window of code generation protocol will open; it contains the information on generation date and time, a directory for output files allocation, and also recordings about the conducted operations (see Fig. 12.5).

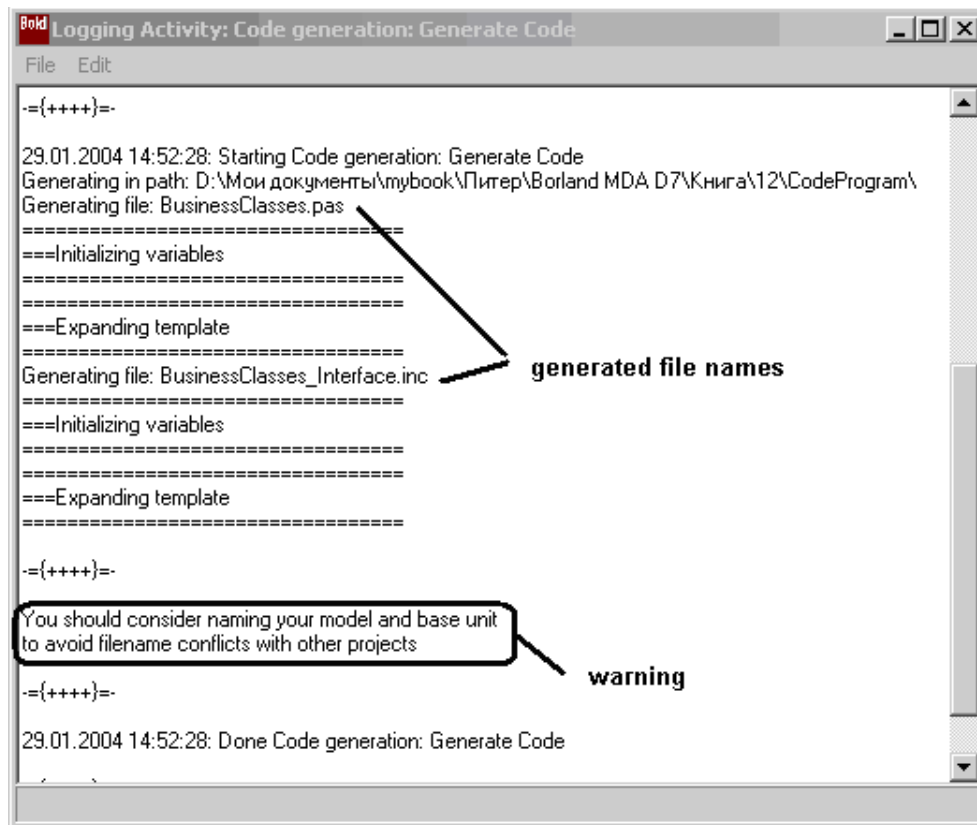


Fig. 12.5. Code generation protocol

At the end of the protocol there is a reminding message regarding necessity of comparison of the names used in model with a name of the main program module, in order to avoid their coincidence, to prevent names conflicts (see Fig. 12.5). As a result of the carried out operation two files are formed on a disk:

- ❑ BusinessClasses_Interface.inc – file of the external program interfaces description;
- ❑ BusinessClasses.pas – file containing UML-model classes code.

At that in Uses section of our project BusinessClasses generated module will be automatically added (see Listing 12.1).

Листинг 12.1. Project program code after generation

```

program Project1;
{%File 'BusinessClasses_Interface.inc'}
uses

```

```

Forms,
Unit1 in 'Unit1.pas' {Form1},
UDataModule in 'UDataModule.pas' {DataModule1: TDataModule},
BusinessClasses in 'BusinessClasses.pas'; // is added automatically
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDataModule1, DataModule1);
  Application.Run;
end.

```

Now we should “give instructions” to used `BoldSystemTypeInfoHandle` system handle of model types about necessity of using the generated program modules for receiving the information on types, for this purpose its "UseGeneratedCode" property should be set in "True" value (see Fig. 12.6).

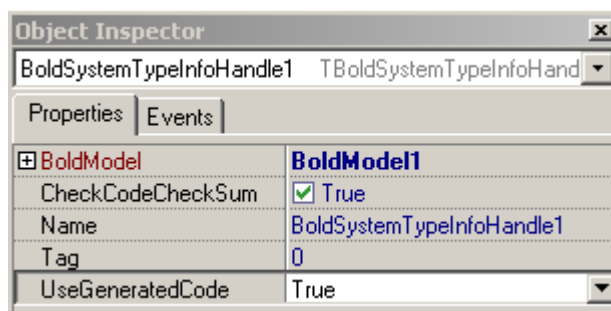


Fig. 12.6. Setting attribute of code using

We should note that if we set the given attribute before code generation, attempt of the application start will call program exception (see Fig. 12.7).

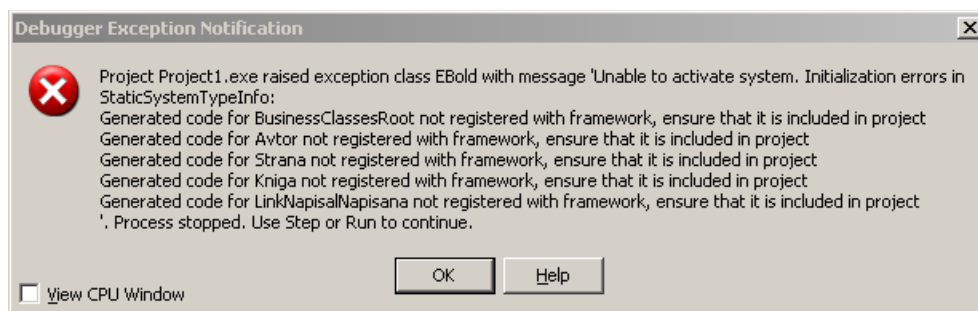


Fig. 12.7. Exception at model code unavailability

Let's try to start our application right away, we shall add several new authors in the list, and we shall make sure that the application functions correctly as before (see Fig. 12.8).

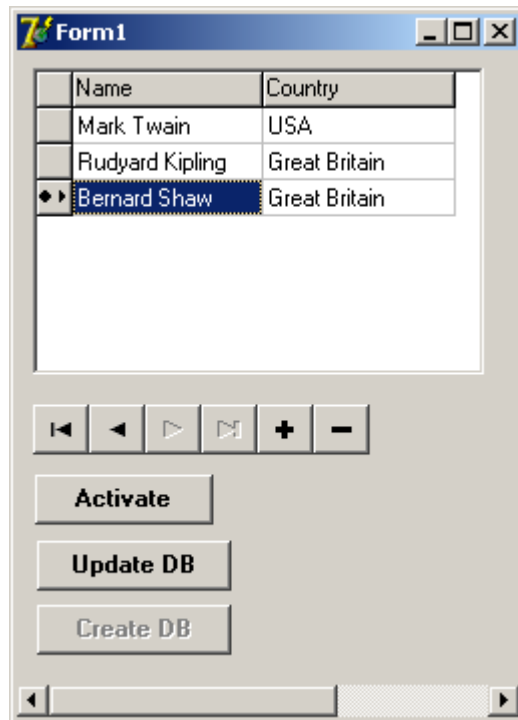


Fig. 12.8. Application run

Generated Modules Structure

Let's examine more in detail program code of model classes, which Bold has generated. We shall start with a file of program interfaces description – `BusinessClasses_Interface.inc`. The fragment of a code from this file is presented below (see Listing 12.2). In the beginning of a code the headline is located; it contains the information on date of file creation, and warning that all "manual" changes brought in this file can be lost. The generated structure of the interfaces description is quite clear, and here we shall pay attention only to some important points. First, it is obvious from listing that for each UML-model class two program classes are generated: one class (for example, `TAuthor`) is intended for model class description, and another class – for the corresponding collection-array (`TAuthorList`) description. Second, associations roles of model class are part of the corresponding program class as classes-properties, and, depending on the concrete role dimension, it can be presented both by object class and by objects collection class.

Listing 12.2. Fragment of interfaces description code

```
*****)
(*      This file is autogenerated      *)
(*  Any manual changes will be LOST!    *)
(*****)
(* Generated 29.01.2004 15:40:13        *)
(*****)
(* This file should be stored in the    *)
(* same directory as the form/datamodule *)
(* with the corresponding model         *)
(*****)
(* Copyright notice:                    *)
(*                                     *)
(*****)

{$IFDEF BusinessClasses_Interface.inc}
{$DEFINE BusinessClasses_Interface.inc}
{$IFDEF BusinessClasses_unitheader}
unit BusinessClasses;
{$ENDIF}
interface
uses
    // interface uses
    // interface dependencies
    // attribute classes
    BoldAttributes,
    // other
    Classes,
    Controls, // for TDate
    SysUtils,
    BoldDefs,
    BoldSubscription,
    BoldDeriver,
    BoldElements,
    BoldDomainElement,
    BoldSystemRT,
    BoldSystem;
type
    { forward declarations of all classes }
    TBusinessClassesRoot = class;
    TBusinessClassesRootList = class;
    TAuthor = class;
    TAuthorList = class;
    TCountry = class;
    TCountryList = class;
    TBook = class;
    TBookList = class;
```

```

TLinkwrotewaswritten = class;
TLinkwrotewaswrittenList = class;
TBusinessClassesRoot = class(TBoldObject)
private
protected
public
end;
TAuthor = class(TBusinessClassesRoot)
private
    function _Get_M_Surname: TBAStrng;
    function _GetSurname: String;
    procedure _SetSurname(const NewValue: String);
    function _Get_M_tpic: TBATypedBlob;
    function _Gettpic: String;
    procedure _Settpic(const NewValue: String);
    function _GetWrote: TBookList;
    function _GetLinkwrotewaswritten: TLinkwrotewaswrittenList;
    function _GetCountry: TCountry;
    function _Get_M_Country: TBoldObjectReference;
    procedure _SetCountry(const value: TCountry);
protected
public
    property M_Surname: TBAStrng read _Get_M_Surname;
    property M_tpic: TBATypedBlob read _Get_M_tpic;
    property M_Wrote: TBookList read _GetWrote;
    property M_Linkwrotewaswritten: TLinkwrotewaswrittenList read
_GetLinkwrotewaswritten;
    property M_Country: TBoldObjectReference read _Get_M_Country;
    property Surname: String read _GetSurname write _SetSurname;
    property tpic: String read _Gettpic write _Settpic;
    property Wrote: TBookList read _GetWrote;
    property Linkwrotewaswritten: TLinkwrotewaswrittenList read
_GetLinkwrotewaswritten;
    property Country: TCountry read _GetCountry write _SetCountry;
end;

```

So, for example, in TAuthor class description there is Country property of TCountry type, and property Wrote of TBookList type. Such distinction can be explained by the circumstance that according to UML-model (see Fig. 12.1) the author can live only in one country (role dimension = 1, **TCountry** program class), and, on the other hand, the author can write many books (role dimension > 1, TBookList program class). In such a way when generating code Bold environment provides the exhaustive description of the business-rules incorporated in UML-model, at the level of generated program structures. And, at last, one more feature is present at the code fragment shown in Listing 12.2. When describing attributes (properties of program classes) it is easy to note "double" declarations of properties. So, for example, for "Surname" attribute of "Author" class there are two similar program declarations:

```

property Surname: String read _GetSurname write _SetSurname
property M_Surname: TBAStrng read _Get_M_Surname;

```

The meaning of the first string is clear: Surname string property and corresponding access methods (procedures) – `_GetSurname` for reading, and `_SetSurname` for recording, are described. And what for Bold has generated the second string with the declaration of similar `M_Surname` property? In order to answer this question, we shall pay attention to type of this property - `TBASString`. In other words this property is not text attribute, but as a matter of fact it plays a role of metainformation descriptor, i.e. contains information on types (from here "M" prefix in the identifier). Such "meta-attributes" represent special objects, which carry out the following important basic functions:

- ❑ Control of attributes values loading;
- ❑ Caching of changes and temporary storage of the attribute previous values;
- ❑ Support of the subscribing mechanism;
- ❑ Calculation of derived-attributes.

For simplicity in many cases it is possible to be guided by the following rule: "A" property represents **value** of "M_A" property. For example, for considered property of `<name>` attribute the following method of access to attribute value (the author surname) is true:

```
name:=M_name.AsString
```

Further, at studying contents of another generated file - `BoldClasses.pas`, we shall make sure in correctness of the formulated rule. We shall consider a fragment (see Listing 12.3), which includes the beginning of this file, and "Author" class program realization.

Listing 12.3. Fragment of `BusinessClasses.pas` module generated code

```
*****)
(*      This file is autogenerated      *)
(*    Any manual changes will be LOST!  *)
*****)
(* Generated 29.01.2004 15:40:13        *)
*****)
(* This file should be stored in the    *)
(* same directory as the form/datamodule *)
(* with the corresponding model         *)
*****)
(* Copyright notice:                    *)
(*                                     *)
*****)
unit BusinessClasses;
{$DEFINE BusinessClasses_unitheader}
{$INCLUDE BusinessClasses_Interface.inc}
{ Includefile for methodimplementations }
const
  BoldMemberAssertInvalidObjectType: string = 'Object of singlelink (%s.%s)
is of wrong type (is %s, should be %s)';
{ TBusinessClassesRoot }
procedure TBusinessClassesRootList.Add(NewObject: TBusinessClassesRoot);
begin
```

```

        if Assigned(NewObject) then
            AddElement(NewObject);
        end;
function TBusinessClassesRootList.IndexOf(anObject: TBusinessClassesRoot):
Integer;
begin
    result := IndexOfElement(anObject);
end;
function TBusinessClassesRootList.includes(anObject: TBusinessClassesRoot) :
Boolean;
begin
    result := IncludesElement(anObject);
end;
function TBusinessClassesRootList.AddNew: TBusinessClassesRoot;
begin
    result := TBusinessClassesRoot(InternalAddNew);
end;
procedure TBusinessClassesRootList.Insert(index: Integer; NewObject:
TBusinessClassesRoot);
begin
    if assigned(NewObject) then
        InsertElement(index, NewObject);
end;
function TBusinessClassesRootList.GetBoldObject(index: Integer):
TBusinessClassesRoot;
begin
    result := TBusinessClassesRoot(GetElement(index));
end;
procedure TBusinessClassesRootList.SetBoldObject(index: Integer; NewObject:
TBusinessClassesRoot);
begin;
    SetElement(index, NewObject);
end;
{ TAuthor }
function TAuthor._Get_M_Surname: TBAStrng;
begin
    assert(ValidateMember('TAuthor', 'Surname', 0, TBAStrng));
    Result := TBAStrng(BoldMembers[0]);
end;
function TAuthor._GetSurname: String;
begin
    Result := M_Surname.AsString;
end;
procedure TAuthor._SetSurname(const NewValue: String);
begin
    M_Surname.AsString := NewValue;
end;
function TAuthor._Get_M_tpic: TBATypedBlob;
begin

```

```

    assert(ValidateMember('TAuthor', 'tpic', 1, TBATypedBlob));
    Result := TBATypedBlob(BoldMembers[1]);
end;
function TAuthor._Gettpic: String;
begin
    Result := M_tpic.AsString;
end;
procedure TAuthor._Settpic(const NewValue: String);
begin
    M_tpic.AsString := NewValue;
end;
function TAuthor._Getwrote: TBookList;
begin
    assert(ValidateMember('TAuthor', 'wrote', 2, TBookList));
    Result := TBookList(BoldMembers[2]);
end;
function TAuthor._GetLinkwrotewaswritten: TLinkwrotewaswrittenList;
begin
    assert(ValidateMember('TAuthor', 'Linkwrotewaswritten', 3,
TLinkwrotewaswrittenList));
    Result := TLinkwrotewaswrittenList(BoldMembers[3]);
end;
function TAuthor._Get_M_Country: TBoldObjectReference;
begin
    assert(ValidateMember('TAuthor', 'Country', 4, TBoldObjectReference));
    Result := TBoldObjectReference(BoldMembers[4]);
end;
function TAuthor._GetCountry: TCountry;
begin
    assert(not assigned(M_Country.BoldObject) or (M_Country.BoldObject is
TCountry), SysUtils.Format(BoldMemberAssertInvalidObjectType, [ClassName,
'Country', M_Country.BoldObject.ClassName, 'TCountry']));
    Result := TCountry(M_Country.BoldObject);
end;
procedure TAuthor._SetCountry(const value: TCountry);
begin
    M_Country.BoldObject := value;
end;
procedure TAuthorList.Add(NewObject: TAuthor);
begin
    if Assigned(NewObject) then
        AddElement(NewObject);
end;
function TAuthorList.IndexOf(anObject: TAuthor): Integer;
begin
    result := IndexOfElement(anObject);
end;
function TAuthorList.Includes(anObject: TAuthor) : Boolean;

```

```

begin
    result := IncludesElement(anObject);
end;
function TAuthorList.AddNew: TAuthor;
begin
    result := TAuthor(InternalAddNew);
end;
procedure TAuthorList.Insert(index: Integer; NewObject: TAuthor);
begin
    if assigned(NewObject) then
        InsertElement(index, NewObject);
end;
function TAuthorList.GetBoldObject(index: Integer): TAuthor;
begin
    result := TAuthor(GetElement(index));
end;
procedure TAuthorList.SetBoldObject(index: Integer; NewObject: TAuthor);
begin
    SetElement(index, NewObject);
end;

```

This fragment also contains headline with the information on date and time of code generation and warning about possible loss of an "extraneous" code. In the beginning of a code there are general methods-procedures of TbusinessClassesRootList class, which in this case represents itself as a super class of model elements. These methods provide operations of objects addition, the current object index gaining, and some others (see Listing 12.3). Further there is the realization code for "Author" and "AuthorList" classes. Right away we should note using identical program structures of a kind:

```

function TAuthor._GetSurname: String;
begin
    Result := M_Surname.AsString;
end;

```

for access to class attributes (program properties) value; we have spoken about this earlier when considering properties such as "M_Surname". In such a way operation of recording in properties-attributes is carried out:

```

procedure TAuthor._SetSurname(const NewValue: String);
begin
    M_Surname.AsString := NewValue;
end;

```

The presented fragments (see Listings 12.2, 12.3) can be useful for independent studying principles of programming of work with objects in Bold for Delphi environment. Probably, it makes no sense to discuss in detail all program structures used in the represented code, in view of code large volume. It will be much more useful to take a closer look at practical implementation of the generated program code when solving specific tasks; and we shall do it in the following section.

Practical Using a Model Code

Work with Classes and Attributes

For the beginning we shall pose a simple task: addition of new authors into the list. For this purpose we shall add Button1 on the form, having given it “Add authors” heading, and into procedure of processing "OnClick" event of this button we shall enter one code string

```
TAuthor.Create(nil);
```

This operator simply creates a new object instance of "TAuthor" type. We shall remind that the identifier of this type is set by "DelphiName" tagged value (see Chapter 4), and by default is formed by addition of “T” prefix to the class name.

Let's start the application run and we shall make sure that new empty strings are added into the authors list by pressing the button (see Fig. 12.10).

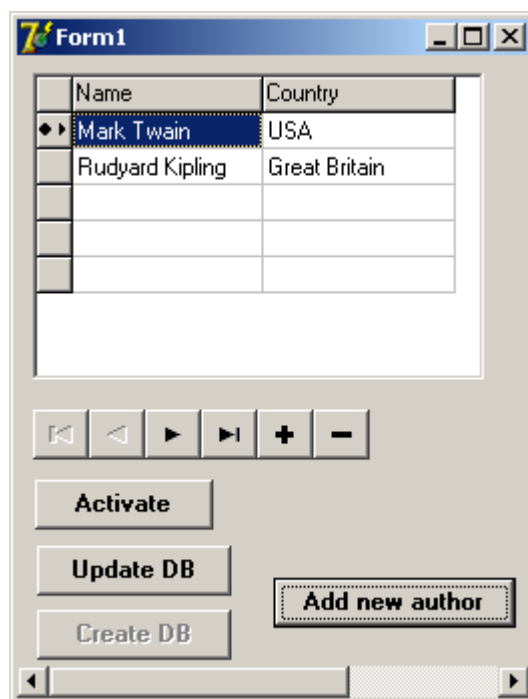


Fig. 12.10. Program addition of “empty” authors

For providing program formation of the author attribute-name we shall add Edit1 input window on the form, into which we shall enter the new author name. We shall modify program response to pressing the button as follows:

```
procedure TForm1.Button4Click(Sender: TObject);
```



```

var NewAuthor:TAuthor;
begin
  if Edit1.Text<>' ' then
  begin
    NewAuthor:=TAuthor.Create(nil);
    NewAuthor.name:=Edit1.Text;
  end;
end;

```

Here we can see that access to class attributes is provided in a simple and natural way. "NewAuthor. name" usual dot-notation accepted in Object Pascal language is used. We shall start the modified application run and we shall make sure that names of added authors really coincide with the text entered into the input window (see Fig. 12.11). If we leave an input window empty, the new author addition will not take place, as this check is present in the event handler written by us.

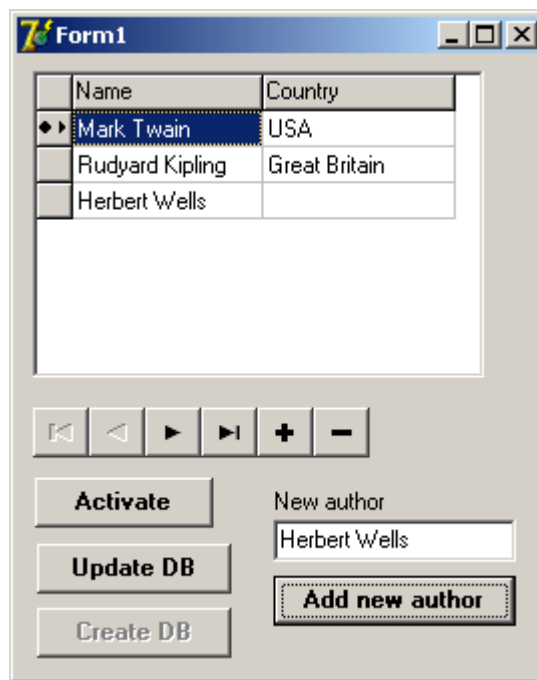


Fig. 12.11. Program assignment of attribute-name

For objects deletion "Delete" method is used. For example, we can delete the new author right after its creation, having used for this purpose the following simple operator:

```
NewAuthor.Delete;
```

ATTENTION

It is impossible to apply a usual "Free" method for operation of model classes objects deletion. The call of the given method will lead to an initiation of program exception.

Work with Associations

Let's complicate a little our application for solving the following task: to map in program mode the country, in which the current author lives. For this purpose we shall add Label1 on the application form of for mapping the country name; and also we shall add one more button – in its pressing handler procedure we shall enter the following program code:

```
procedure TForm1.Button5Click(Sender: TObject);
var CurAuthor:TAuthor;
begin
    CurAuthor:=ListAllAuthors.CurrentBoldObject as TAuthor;
    Label1.Caption:=CurAuthor.Country.Name;
end;
```

Once more time we can see that program access to the association and its role from a program code is provided simply and obviously. “CurAuthor.Country.Name” operator is completely “transparent” and looks very simple, however at that it realizes rather complex operation of access to other class ("Country") and obtaining "Name" value-attribute from it. The first operator in the above-stated code fragment has also simple sense – to set value of the current author in the list to TAuthor object. After the application start we shall make sure that the posed task is solved (see Fig. 12.12).

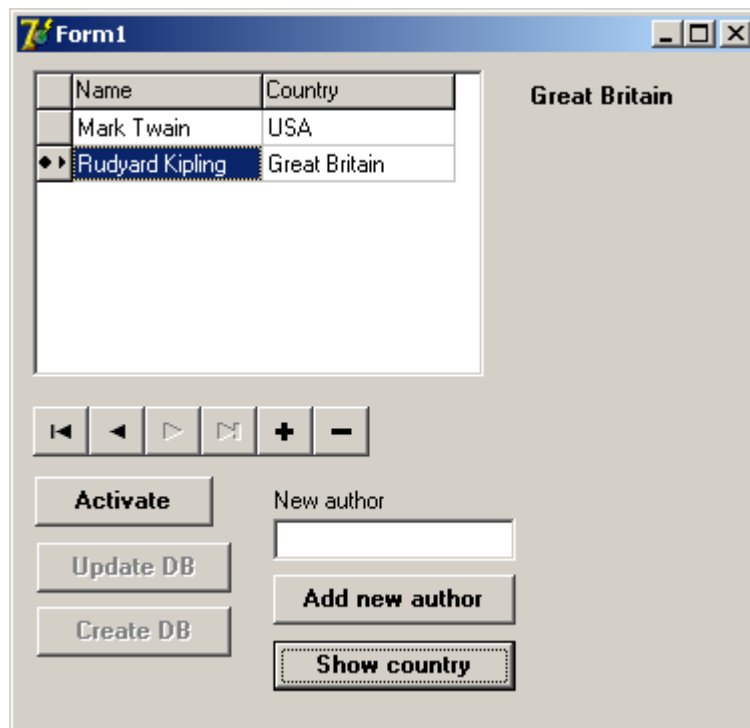


Fig. 12.12. Access to the association role value

Now, in the same simple way, when adding the new author, we shall try to assign him the country, where he lives. Let's add in "Add author" button event handler the following operator:

```
NewAuthor.Country.Name:='USA';
```

As a result after attempt to add the new author we shall receive program exception (see Fig. 12.13), which message box notifies that TCountry class does not exist.



Fig. 12.13. The attempt of access to the non-existent object

It is quite obvious result, as when adding the new author, he does not have links with any country. We shall try to create the author and the country of residing simultaneously. The name of the country we will enter into the second input window – Edit2. For the best understanding we shall place on the form countries list handler - ListAllCountry and the second grid - BoldGrid2 for mapping this list of the countries. For the country creation we shall change response on pressing "Add author" button:

```

procedure TForm1.Button4Click(Sender: TObject);
var NewAuthor:TAuthor;
    NewCountry :TCountry;
begin
    if Edit1.Text<>' ' then
    begin
        NewAuthor:=TAuthor.Create(nil);
        NewAuthor.name:=Edit1.Text;
        NewCountry:=TCountry.Create(nil);
        NewCountry.Name:=Edit2.Text;
        NewAuthor.Country:=NewCountry;
    end;
end;

```

Let's start the application run and add the new author 'Pushkin' and country 'Russia' (see Fig. 12.14).

Fig. 12.14. Simultaneous creation of two objects and links between them

It may seem that the posed task is solved. However, this is not the case. After input of the second author living in the same country, we shall find out that the same country (Russia) is repeated twice in the countries list (see Fig. 12.15). It occurs because every time, when we

automatically add new object-country, we don't check, whether such country is already present in the list.

Fig. 12.15. Incorrect duplication of the objects-countries

We shall complicate our handler procedure of pressing the button (see Listing 12.4). When adding the new author, procedure checks, whether there is in the list of the countries a country with the name entered into Edit2 input window. If the country exists, it gets into touch with the new author. Otherwise the new object-country is created, and it also contacts with the new author. The procedure code is supplied with detailed comments (see Listing 12.4).

Listing 12.4. Procedure of adding new author and new country of residing, if necessary

```
procedure TForm1.Button4Click(Sender: TObject);
var NewAuthor :TAuthor;
    CurCountry :TCountry;
    i: integer;
    bCountryExists :boolean; // attribute of country existence
begin
  if Edit1.Text<>'' then
  begin
    NewAuthor:=TAuthor.Create(nil); // new author creation
    NewAuthor.name:=Edit1.Text;      // author name, taken from Edit1
```

```

        bCountryExists:=False;           // we assume that the country does not
exist
// A cycle on all elements of the countries list
    for i:=0 to ListAllCountry.Count-1 do
        begin
            CurCountry:=ListAllCountry.ObjectList.Elements[i] as TCountry;
//current // element-country
            if CurCountry.Name=Edit2.Text then // check of the name coincidence //
of the current country and the text, entered in Edit2
                begin
                    bCountryExists:=true; // we set the attribute of the country
existence
                    break; // loop exit, if we have found the country in the list
                end;
            end;
        if (not bCountryExists) then // if the country is not found, we create
the new one
            begin
                CurCountry:=TCountry.Create(nil); // we create the new object-country
                CurCountry.Name:=Edit2.Text; // country name from Edit2
            end;
            NewAuthor.Country:=CurCountry; // country assignment to the new author
        end;
    end;
end;

```

After the application start we make sure that the posed task is solved.

Operations

The code generation allows using *the operations* included in model classes. As a matter of fact, operations represent analogues of methods in object-oriented programming. For this reason operations formed in UML-model at code generation "are mapped" into usual classes methods. Each operation can have input parameters and return the certain type, i.e. to play a role of functions or procedures. We shall consider by the simple example, how to realize the operations manipulation. We shall create BookCount operation for calculation of total of the books written by the author. To complete the picture we shall create operation in Rational Rose. For this purpose we shall load our model in Rational Rose class diagrams editor (see Chapter 4) and, having called "Author" class specification, we shall pass on "Operations" bookmark (see Fig. 12.17).

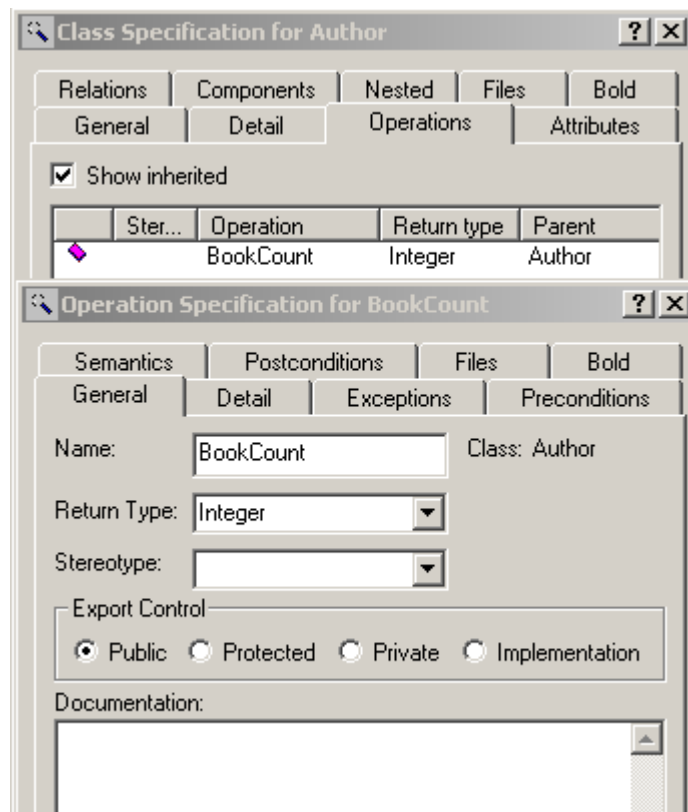


Fig. 12.17. Adjustment of operation properties in Rational Rose

Having called a pop-up menu, we shall add operation, and after double click on it we shall pass to the window of new operation specification. We shall give BookCount name to the operation and set integer type of returned result (see Fig. 12.17). After that the added operation will appear on the class image in UML-model (see Fig. 12.18).

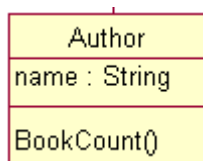


Fig. 12.18. Class with BookCount operation

We shall import the advanced model into Bold for Delphi environment, and check whether operation is mapped in model elements tree (see Fig. 12.19).

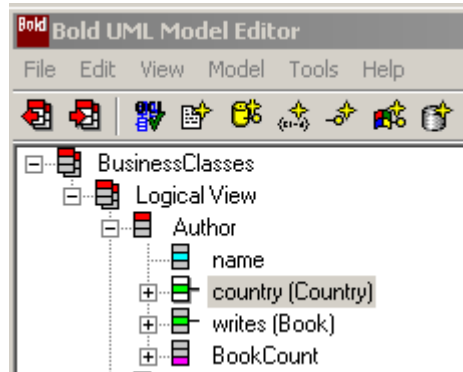


Fig. 12.19. Model tree with the added operation

Now we should generate code of model classes once again, in order that the added new element of UML-model has represented in it. When we carry out this operation, we shall find out that at generation the new file with "BusinessClasses.inc" name appears; it contains program template for BookCount operation (see Listing 12.5).

Listing 12.5. Program code template for new operation

```

*****
(*)
(*) Bold for Delphi Stub File (*)
(*)
(*) Autogenerated file for method implementations (*)
(*)
(*****)
//
{$INCLUDE BusinessClasses_Interface.inc}

function TAuthor.BookCount: Integer;
begin

end;
```

Now we have to enter necessary operators into the function body. In this case, for calculating total of books it is enough to write only one string of a program code:

```
Result:=Writes.Count;
```

This operator will provide return of total amount of "Writes" role values, which corresponds to quantity of books written by the author. It is necessary to note again simplicity and the elegance of program expressions provided by Bold environment when using code generation.

For checking operation functioning we shall add on the form Label1 standard label and the button named "Operation", into which pressing event handler we shall enter the following code:

```
procedure TForm1.Button4Click(Sender: TObject);
var CurAuthor:TAutor;
begin
    CurAuthor:=ListAllAuthors.CurrentBoldObject as TAuthor;
    Label1.Caption:='Number of books  '+IntToStr(CurAuthor.BookCount);
end;
```

We see that the operation reference is realized in the same way, as reference to any other class method – “CurAuthor.BookCount”. After starting the application we shall make sure that the operation perfectly carries out books calculation depending on the selected author .

Naturally, such simple task could be solved much easier with the help of OCL and one derived attribute. But it is not the reason to underestimate operations, because, nevertheless, OCL capabilities are limited, but program code capabilities practically have no constraints, therefore with the help of operations in practice it is possible to encapsulate in model classes and realize by program approach very complex methods of controlling objects behaviour and information processing.

Summary

Code generation is convenient means for creation of the program objects reflecting UML-model elements. It is possible to work with these program objects, just as with usual classes in OOP, at that access methods concerning attributes, associations and roles present in model are realized very easily and evidently. One more advantage when applying the code generator is the possibility of creation and use of operations, which in the generated program code are mapped in usual classes methods. In the following Chapter other advantages of code generation use will be shown.

Chapter 13. Subscribing Mechanism

In this chapter the so-called subscribing mechanism will be considered. Literally, the given mechanism plays the dominant role in the organizing interoperability of components and layers of the application during its run. Synchronization of GUI and business layer, derived attributes and associations, and realization of many other functions is provided by subscribing mechanisms.

The Description and Realization

Events and Subscription

In one of the previous chapters it was mentioned that, as a matter of fact, Bold for Delphi program system represents “system in system”. Possessing own memory manager, own types-expansions of standard types, Bold for Delphi also possesses own means for messages exchange. Such messages occur, for example, when any object attribute status changes, or there is a moving in the objects list, etc. Thus, messages are generated by Bold environment, when the certain events take place. All events can be divided on internal and user-defined. Internal events are used by the program environment. For example, synchronization of GUI elements is provided due to the numerous internal events taking place at business-layer, when changing the status of any object space elements. The user-defined events are intended for realizing by the developer. However, it is obvious that only events presence is not enough for solving problems of interoperability between system elements. Presence of some mechanism, which provides not only generation of events, but also their tracking and support of responses realization, is necessary. The subscribing mechanism acts as such tool. This mechanism enables "to subscribe" a concrete element on the certain event taking place in system. At that it is possible to subscribe on absolutely concrete event, for example, on changing the status of concrete attribute of a concrete class object. And, for example, at changing the value of such attribute by the user, the element-"subscriber" will receive the notice from system, and realize some response. If we look at functioning Bold-application during its run, we can imagine some "living" system consisting of set of elements, which constantly "communicate" among themselves, developing messages-events, and, in turn, "responding" to these messages. Simple click a mouse button on a grid causes quick "revival" of such "community", however, it is not chaotic, but absolutely ordered and controllable by software system kernel. Once again we shall emphasize that events in Bold for Delphi, and usual events in Windows are different things, and the subscribing mechanism is realized by Bold for Delphi program system quite independently.

NOTE

Probably, one of the reasons of such "isolation" of event environment in Bold for Delphi product was intention of developers to port further this product on other operating systems. Unfortunately, at present time it is not realized yet.

The Basic Classes and Realization

The possibility of using the subscribing mechanism "is build" by Bold for Delphi developers into the important internal class (see Chapter 6) – **TBoldElement**, which is a super class for any types of object space elements. Direct "parent" of **TBoldElement** class is **TBoldSubscribableObject** class; its role is clear from its name – this class is an abstract class for all those child types, which "require" the subscription use. Direct "implementators" of the subscribing mechanism are the following classes: **TBoldPublisher** (included into **TBoldSubscribableObject** class as the property) and **TBoldSubscriber**. **TBoldPublisher** class provides addition of subscribers by **AddSubscription** and **AddSmallSubscription** methods. Difference of these two methods is the following. The concept of "small", or it is better to say, standard subscription, uses the standard events reserved by Bold developers. Each event is described by simple integer-number, and for standard events numbers from zero up to 31 are selected (see Table 13.1). From this range of events numbers from 24 up to 31 are assigned for the user "small" events, which also can be used in standard subscriptions.

Table 13.1. List of the basic standard events

Number (value)	Name	When it is produced
0	beDestroying	Before destroying the object
2	beMemberChanged	It generates the object, when changing its member (attribute)
3	beObjectDeleted	After deleting the object
4	beItemAdded	When adding the object into the list
5	beItemDeleted	When deleting the object from the list
6	beItemReplaced	When changing the object in the list onto another object
7	beOrderChanged	When changing the elements order in the list
8	beValueChanged	It generates BoldElement (for example attribute) after changing its value
11	beObjectCreated	When creating the new object
12	beValueInvalid	When the element value is invalid
17	beDeactivating	When deactivating persistence layer
18	beRolledBack	It generates BoldSystem when transaction is rolled back
21	beObjectFetched	It sends the object when it is fetched

An advantage of using a standard subscription (**AddSmallSubscription** method) is that in one request for subscription it is possible to subscribe at once for several events (though it is achieved by the above mentioned constraint of standard events set). Functioning of the subscribing mechanism occurs as follows. The element, "wishing" to subscribe to some event, calls **TBoldPublisher** class methods (this class is present in many standard classes, because most of used classes are **TBoldElement** successors), **AddSmallSubscription** or **AddSubscription**, using object of **TBoldSubscriber** type as a parameter. Besides, request-

parameter hands over the number of event (or several numbers simultaneously, for standard subscription by AddSmallSubscription method) and the mandatory event identifier. The identifier is used by the mechanism for binding to the subscriber. Further, if the given event has come, BoldPublisher object checks, whether there are subscribers to this event, and for all found subscribers calls Receive procedure – TboldSubscriber class method (the object of this class is passed as parameter at subscription "registration"), in which the developer can place response to the given event. Below in this chapter it will be considered by a concrete example, how the aforesaid is carried out in practice. This is very simplified description of the mechanism, in which actually much more capabilities are incorporated.

And, though the subscribing mechanism is rather large and is very difficult for perception, the situation is essentially eased by the fact that in practice necessity of "manual" use of the given mechanism occurs seldom. It can be explained by the following: basically the subscribing mechanism is used by Bold for Delphi program environment, which provides the developer with well adapted high-level means for realization of the overwhelming majority of required capabilities. At the end of this chapter we shall examine examples of work with such means: program work with derived attributes, and also using the so-called ReverseDerived attributes, which presence is a unique capability of Bold for Delphi product.

Subscription Program Use

In this section the basic methods of work with the subscribing mechanism using "manual" programming will be illustrated by a concrete example. As a basis we shall use the simple applications mapping the list of authors, which were created in the previous chapters. When working with this list, the user can make various operations: addition, deletion, changing the records, etc. At that in system corresponding events are automatically developed. We shall show by a simple example how "to subscribe" for such events, and to process corresponding responses, when receiving notices on such events occurrence. For subscription it is necessary at least, two components to be present, - the one who subscribes, and that, "to whom the subscription is made". We shall subscribe for the list of authors, which is presented by ListAllAuthors list handle. And actually "subscriber" will be created by us. We shall create the class-successor of TboldSubscriber class as such element. For this purpose we shall add in the program module the new class description, having named it TNewSubscriber:

```
type
  TNewSubscriber=class(TboldSubscriber)
end;
```

Let's remind that (after further correct "registration" of our subscription) when the event occurs (it is not clear what event, as we have not chosen it yet), Bold environment will automatically call Receive method of our subscriber. In TboldSubscriber parental class this procedure is not filled with the contents, as TboldSubscriber is an abstract class. Therefore it is necessary to redefine Receive standard procedure that can be made by simple addition of a string (the procedure code will be written later):

```
type
  TNewSubscriber=class(TboldSubscriber)
  procedure Receive(Originator: TObject);
```

```
OriginalEvent: TBoldEvent; RequestedEvent: TBoldRequestedEvent);override;
end;
```

Let's consider more in detail structure of Receive procedure parameters:

- ❑ Originator – specifies the event source;
- ❑ OriginalEvent – type of the come event (its number); due to this parameter it is possible to define what event has come;
- ❑ RequestedEvent – the event identifier necessary for binding of subscribers to the concrete subscription.

All listed parameters of Receive procedure are input parameters, and they are formed automatically by program environment (by object of TBoldPublisher type of the element, on which the subscription is carried out). Further, for use in the program, we should describe variable-object of the type created by us. We shall name it NewSub.

```
Var NewSub : TNewSubscriber;
```

Besides we shall set for the subscriber any identifier, for example MyEvent

```
Const MyEvent=58;
```

Let's start registration of our subscription. We shall place on the form "Subscribe" button, and enter the following code of response to its click:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
    NewSub:=TNewSubscriber.Create;
    ListAllAuthors.List.AddSmallSubscription(NewSub,[beItemAdded],MyEvent);
end;
```

The first operator creates an instance of NewSub subscriber (which should be deleted later by NewSub.Free usual method). The second operator forms a standard subscription, specifying as NewSub subscriber parameters a set of standard events (consisting of one beItemAdded event), and MyEvent identifier. In this case we have used the situation that in ListAllAuthors list class-handle there is already a property of TBoldPublisher type allowing us to use AddSmallSubscription method. beItemAdded event used in the subscription occurs when adding an element in the list. Now we need to realize program response to the given event. For this purpose we shall create Receive procedure code for TnewSubscriber class-subscriber:

```
Procedure TNewSubscriber.receive;
begin
    ShowMessage('Attention! The author is added into the list!');
end;
```

That is, after the message comes the given procedure should call occurrence of a standard dialogue box. We shall start the application run, and after pressing "Subscribe" button we shall try to add the new author (see Fig. 13.1).

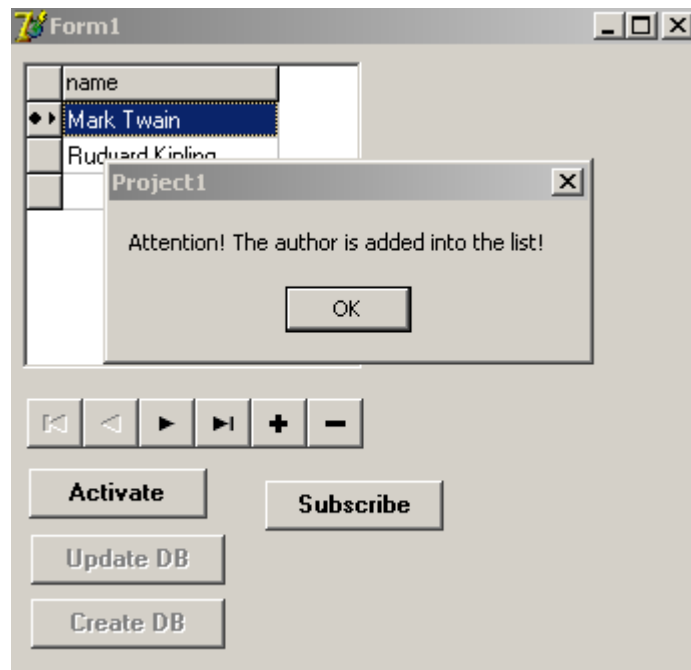


Fig. 13.1. Response to event

We shall make sure that the subscribing mechanism works. If we try to delete or edit authors, nothing will take place. Now we shall modify our application a little. First, we "shall subscribe" simultaneously for two events: authors addition (`beItemAdded`) and deletion (`beItemDeleted`).

```
procedure TForm1.Button5Click(Sender: TObject);
begin
    NewSub:=TNewSubscriber.Create;
    listallauthors.List
        .AddSmallSubscription(NewSub,[beItemAdded,beItemDeleted],MyEvent);
end;
```

Second, we shall try to distinguish, which event has taken place, in Receive procedure:

```
Procedure TNewSubscriber.receive;
begin
    case OriginalEvent of
        beItemDeleted: ShowMessage('Attention! The author is deleted from the
list!');
        beItemAdded:   ShowMessage('Attention! The author is added into the
list!');
    end; // case
end;
```

After the application start we shall make sure that different messages are displayed as a result of addition or deletion of authors .

Based on the considered example it is possible if necessary to create more complex variants of the subscription.

Using BoldPlaceableSubscriber

In the previous chapter we have already applied BoldPlaceableSubscriber component, but not quite on its direct purpose. Actually this component is intended for generating user subscriptions, at that lightening significantly their realization, and eliminating necessity of "manual" creation of classes and procedures, as we did it in the previous example. We shall consider how to work with BoldPlaceableSubscriber component. For this purpose we shall place a component on the form, and pay attention to its properties-events. There are two properties-events - OnReceive and OnSubscribeToElement. Purpose of OnReceive event handler is equivalent to Receive procedure described above, i.e. in this handler procedure of response to the subscribed event is located. The second handler, for OnSubscribeToElement event, allows "to register" the subscription. Let's pose the following task - to trace work with the database regarding addition and deletion of books for concrete authors. At first we shall connect BoldPlaceableSubscriber component to the authors list (BoldHandle property). Further, we shall register our subscription, having entered into OnSubscribeToElement event handler the following code:

```
procedure TForm1.BoldPlaceableSubscriber1SubscribeToElement(
  element: TBoldElement; subscriber: TBoldSubscriber);
begin
  Element.SubscribeToExpression
    ('Author.allInstances->select(name='Pushkin')->first.writes',
  subscriber, true);
end;
```

In this handler the property inherent in all OS elements (successors of TBoldElement class) is used – the capability to subscribe for OCL-expression result. In this case we subscribed on number of the books, written by concrete author (Pushkin), involving OCL-navigation on model

```
'Author.allInstances->select(name='Pushkin')->first.writes'
```

What does “the subscription to OCL-expression result” mean? It means that at any change of OCL-expression evaluation result the subscriber will automatically receive the notice, i.e. as we have already known, Receive method will be called (in this case – OnReceive handler). Into OnReceive event handler we shall enter the following code:

```
procedure TForm1.BoldPlaceableSubscriber1Receive(
  sender: TBoldPlaceableSubscriber; Originator: TObject; OriginalEvent,
  RequestedEvent: Integer);
begin
  if OriginalEvent=beItemAdded then
    if (Originator is TBookList) then ShowMessage(' Attempt to add the book to Pushkin
author!');
  if OriginalEvent=beItemDeleted then
```

```

    if (Originator is TBookList) then ShowMessage(' Attempt to delete the book from
Pushkin author!');
end;

```

In this handler it is checked, which event has come, and who is its source. Originator object, which value is compared with TBookList class acts as a source of event. This check is necessary to exclude from processing the events occurring at deletion or addition of authors, as in this case we are interested in books, but not in authors. If now we start the application, it is easy to make sure that responses to events take place only when manipulations (addition or deletion) with books of the concrete author - "Pushkin" - are made. In all other cases (addition or deletion of authors, or additions and deletions of books to (from) other authors), the program will not give a response. Thus, we can be convinced once again that Bold for Delphi program environment provide us with perfect flexible capabilities for "interception" of concrete events.

Programming Derived Attributes

Earlier in this book we have already got acquainted with derived attributes (see Chapter 8). They are analogues of derived fields of tables in the traditional scheme of databases applications development. For work with derived attributes in Bold for Delphi environment it is very convenient to use capabilities of OCL for specifying their values calculating rules. However in practice there can be situations when for these purposes there is not enough OCL capabilities. And then the only way to solve such problems is using a program code. Fortunately, Bold for Delphi developers have provided such variant of work with derived attributes beforehand. Below we shall consider, how programming values of derived attributes is realized in practice. Why do we examine this subject in this chapter? The matter is that, as it will be shown below, methods of derived attributes programming are connected directly to subscribing mechanisms considered in the given chapter. It is necessary to note just here that using a program code for forming derived attributes values requires mandatory code generation (see the previous Chapter). We shall consider the following task. Let for the account of some goods in a database it is necessary to carry out automatic binding of the moment of inputting the record about each new goods to the current date and the moment of time. For solving the posed task we shall create the simple class containing the name of the goods, and two derived attributes for the automatic description of date and time (see Fig. 13.3). How in this case to generate expressions for derived attributes? After all, in OCL there are no operations for receiving date and time.

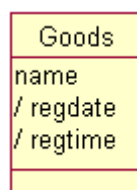


Fig. 13.3. Class with derived-attributes

In this situation there is the following way out – to use program calculation for receiving values of derived attributes. For this purpose first of all it is necessary to generate a model code (see Chapter 12). After code generation BusinessClasses.inc file is formed in addition;

it contains program templates of procedures for calculating values of the current date and time (see Listing 13.1).

Listing 13.1. Code Generated for Derived-attributes

```
(*****)
(*)
(*) Bold for Delphi Stub File (*)
(*)
(*) Autogenerated file for method implementations (*)
(*)
(*)
(*****)

//
{$INCLUDE BusinessClasses_Interface.inc}
procedure TGoods._RegDate_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
//var
// Result: String;
begin
    // Calculate value into Result and place the required subscriptions
    // Result := <<formula>>
    // M_RegDate.AsString := Result;
end;
procedure TGoods._RegTime_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
//var
// Result: String;
begin
    // Calculate value into Result and place the required subscriptions
    // Result := <<formula>>
    // M_RegTime.AsString := Result;
end;
```

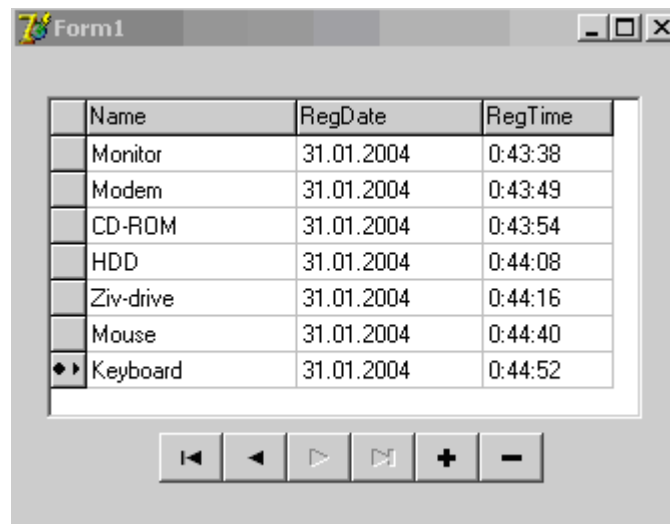
The Listing is supplied with the necessary comments, which help the developer to write program expressions. In this case these expressions will look simply enough (see Listing 13.2).

Listing 13.2. Calculating Derived-attributes in Program

```
procedure TGoods._RegDate_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
begin
    M_RegDate.AsString := DateToStr(Date);
end;
procedure TGoods._RegTime_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
begin
    M_RegTime.AsString := TimeToStr(Time);
```

end;

Let's create the simple GUI for checking workability of the application, and we shall make sure that, when entering the new goods, date and time of input are formed automatically (see Fig. 13.4).



	Name	RegDate	RegTime
	Monitor	31.01.2004	0:43:38
	Modem	31.01.2004	0:43:49
	CD-ROM	31.01.2004	0:43:54
	HDD	31.01.2004	0:44:08
	Zip-drive	31.01.2004	0:44:16
	Mouse	31.01.2004	0:44:40
▶	Keyboard	31.01.2004	0:44:52

Fig. 13.4. Automatic registration of the record input moment

However we have not completely solved the task yet, because values of derived attributes are not saved at the persistence layer. In order to make sure in this fact we shall simply close the application and start it again. We shall see that all values generated earlier will be automatically replaced with the current date and the moment of the program restart (see Fig. 13.5).

Name	RegDate	RegTime
Keyboard	31.01.2004	0:45:32
Mouse	31.01.2004	0:45:32
Ziv-drive	31.01.2004	0:45:32
HDD	31.01.2004	0:45:32
CD-ROM	31.01.2004	0:45:32
Modem	31.01.2004	0:45:32
Monitor	31.01.2004	0:45:32

Fig. 13.5. Derived-attributes are not saved in DB

However it is rather easy to improve the considered application with the purpose of saving the derived values. We shall show how to do it in a simple way. We shall add in Goods class two usual text attributes, which we shall use for storage of Derived-attributes values (see Fig. 13.6).

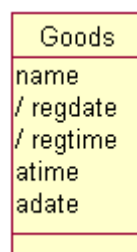


Fig. 13.6. Adding persistent-attributes

We shall change a little the text of procedures of calculating the fields (see Listing 13.3).

Listing 13.3. Saving the Values of Derived-attributes

```

procedure TGoods._RegDate_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
begin
    M_RegDate.AsString := DateToStr(Date);
  
```

```

    if (aDate='') then M_aDate.AsString:=RegDate;
end;

procedure TGoods._RegTime_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoIdSubscriber);
begin
    M_RegTime.AsString := TimeToStr(Time);
    if (aTime='') then M_aTime.AsString:=RegTime;
end;

```

Values of derived attributes are given to usual attributes, however it is necessary to check beforehand that usual attribute is empty (i.e. it belongs to new object). If we don't do such check, it is obviously that the previous situation will be repeated. For clearness we shall map in a grid all object attributes and enter several values. After that we shall update a database and restart the application (see Fig. 13.8).

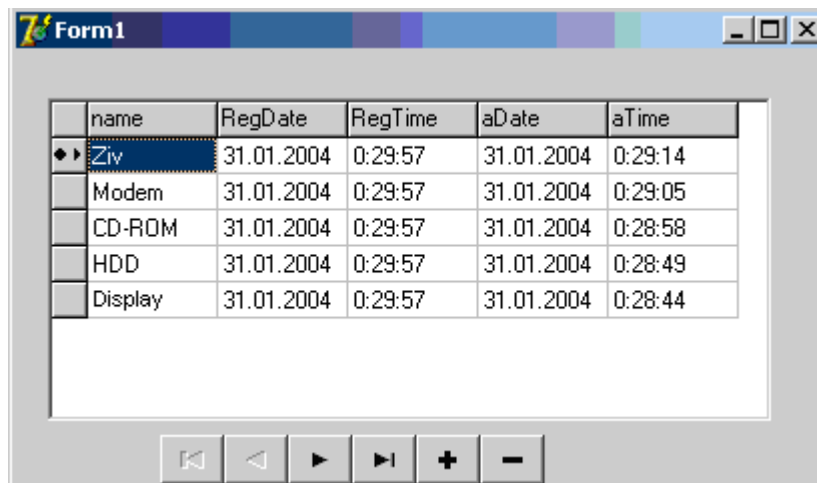


Fig. 13.8. The derived values instances are saved in DB

We can see that at restart derived attributes were copied anew, but the former information was saved in their attributes - "doubles". Thus, the posed task can be considered to be solved.

NOTE

The described method of saving values of derived attributes in a database can be expanded and apply in practice in many cases, as such tasks occurs fairly often in real development.

Summing up, it is possible to ascertain that using a program code for forming derived attributes essentially expands possibilities of their implementation, allowing realizing the functionality unavailable for OCL-expressions.

Reverse-derived Attributes

Reverse-derived attributes are unique feature of Bold for Delphi environment. Usual derived attributes form the values inaccessible for editing. And it is quite understandable. For example, it is possible to use derived attribute for calculating total salary of the department employees. In this case "manual" editing of this attribute value would lead to ambiguity – really how can we “calculate back” the salary of each employee from the received total sum of employees’ salaries? In fact the given attribute should map the sum of salaries. However there are some situations when such “reverse derivations” are possible. For example, from the full string “Roosevelt Franklin Delano” received in result of pasting together a surname, name and patronymic it is simple enough to retrieve these parameters. This circumstance is used by Bold for Delphi environment, allowing the developer to use such “reverse derivations”, when it is possible. We shall consider this interesting capability by the example of solving the following task. We shall assume that the goods database automatically imports the information on dimensions of each goods from invoice, as a string such as "100x200x300", where goods dimensions (length, width and height in millimeters) are specified with “x” separating symbol between them. It is necessary to provide data reconstruction on length, width and height of the goods, at that with a capability of automatic calculation of the goods volume in liters, and also with a capability of manual editing both a full string of dimensions, and each parameter taken separately. We shall show how easy we can solve this task using “reverse derivations” in Bold for Delphi. For the beginning it is necessary to specify structure of "Goods" class attributes. We shall add L, D and H integer attributes, corresponding to length, width and height. Also we shall add V actual derived attribute (automatically derived volume of the goods), and, at last, a string of dimensions – **Dimensions attribute**. This attribute also should be derived, because according to the problem situation it should be formed automatically at manual editing length, width or heights. We shall import received in such a way model, consisting of one class, into Bold for Delphi environment and start the built-in editor of models. In the models editor we shall set “Reverse derive” property for “**Dimensions**” attribute.

It will allow us to retrieve length, width and height from the complete string of the dimensions, presented by this attribute. Further, we shall make operation of code generation. BusinessClasses.inc file formed in result will contain templates for realization of program methods, as well as earlier (see Listing 13.4).

Listing 13.4. Generated code, including Reverse-derived attribute support

```
procedure TGoods._Dimensions_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoldSubscriber);
//var
// Result: String;
begin
    // Calculate value into Result and place the required subscriptions
    // Result := <<formula>>
    // M_Dimensions.AsString := Result;
end;
procedure TGoods._Dimensions_ReverseDerive(DerivedObject: TObject);
begin
```

```

end;
procedure TGoods._V_DeriveAndSubscribe(DerivedObject: TObject; Subscriber:
TBoIdSubscriber);
//var
// Result: double;
begin
    // Calculate value into Result and place the required subscriptions
    // Result := <<formula>>
    // M_V.AsFloat := Result;
end;

```

But, as against the examples considered above, this code also contains empty program template for realizing reverse **derivations**.

```

procedure TGoods._Dimensions_ReverseDerive(DerivedObject: TObject);

```

Programming we shall begin with usual derived-attributes. For procedure of volume **derivation** we shall write the following code:

```

procedure TGoods._V_DeriveAndSubscribe(DerivedObject: TObject; Subscriber:
TBoIdSubscriber);
begin
    M_V.AsFloat := L/100.0*D/100.0*H/100.0;
    M_L.DefaultSubscribe(Subscriber);
    M_D.DefaultSubscribe(Subscriber);
    M_H.DefaultSubscribe(Subscriber);
end;

```

In the first code string the goods volume converted in liters is derived. The next strings illustrate use of the subscribing mechanism; in this case DefaultSubscribe method is called for each attribute participating in volume **derivation**. It should be done in order that subscriber learn about this event and recount volume at any change of length, width or height. Similar procedure we shall write also for "pasting" dimensions with "x" separating symbol:

```

procedure TGoods._Dimensions_DeriveAndSubscribe(DerivedObject: TObject;
Subscriber: TBoIdSubscriber);
begin
    M_Dimensions.AsString:=IntToStr(L)+'x'+IntToStr(D)+'x'+IntToStr(H);
    M_L.DefaultSubscribe(Subscriber);
    M_D.DefaultSubscribe(Subscriber);
    M_H.DefaultSubscribe(Subscriber);
end;

```

And once again in this procedure we "subscribe" on changing L, D, H parameters for updating, if necessary, the pasted string-attribute – Dimensions. And, at last, we shall write a code for “reverse **derivation**” of length, width and height from dimensions string. This code is separated as "DecodeDimensions" procedure, receiving a "sth" string at input, and returning three integer retrieved parameters - oL, oD, oH, which present length, width and height. We shall note that this procedure does not depend on used separating symbol

between dimensions (the main thing that it is not a numerical value). The procedure text is quite transparent and is presented below:

```
Procedure DecodeDimensions(sth :string; var oL,oD,oH : Integer);
var st:string;
    j : word;
begin
    if st='' then exit;
    st:=trim(sth);
    for j:=1 to length(st) do if NOT (st[j] in ['0'..'9']) then break;
    oL:=strtoint(Copy(st,1,j-1));
    Delete(st,1,j);
    for j:=1 to length(st) do if NOT (st[j] in ['0'..'9']) then break;
    oD:=strtoint(Copy(st,1,j-1));
    Delete(st,1,j);
    oH:=strtoint(st);
end;
```

After that we need to call the given procedure in reverse-derived attribute method and to give values returned by procedure to L, D, H attributes:

```
procedure TGoods._Dimensions_ReverseDerive(DerivedObject: Tobject);
var oL,oD,oH : integer;
begin
    DecodeDimensions(Dimensions,oL,oD,oH);
    L:=oL;
    D:=oD;
    H:=oH;
end;
```

Let's start the application run (see Fig. 13.11). At addition of new record the “pasted” string of dimensions with zero initial values is automatically mapped.

name	L	D	H	V
Gabarits				
0x0x0	0	0	0	0

Fig. 13.11. The new record addition

After that it is possible to work with the application in different ways, and this process is rather flexible. It is possible to enter separately length, width or height, and after such input the volume will be automatically recounted, and also the new string of dimensions will be pasted together and mapped. This is functioning in style of usual derived attributes. And it is possible to act on the contrary: to edit directly dimensions string either in BoldGrid, or having called the autoform (see Fig. 13.12). After editing dimensions string the length, width and height will automatically be updated, and also the goods volume will be automatically recounted.

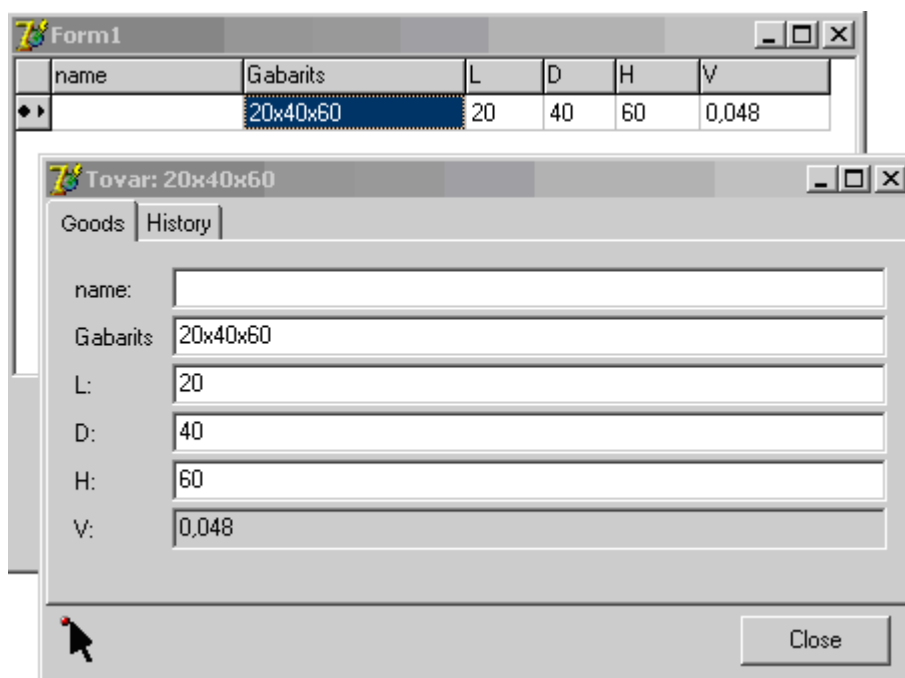


Fig. 13.12. Editing the reverse-derived attribute

Such way of editing is qualitatively new, and it is possible realize it due to reverse-derived attributes. As a result of their use we have completely solved the posed task , having received at that the application with very flexible interface.

Summary

The subscribing mechanism to events is built in Bold for Delphi program system, and it is actively used by the environment for controlling interoperability of elements in process of the application run. Synchronization of the application layers (a business-layer, the GUI and persistence layer) is provided with subscribing mechanisms. **Calculation** of derived-attributes and OCL-expressions evaluation are also in many respects provided by means of these mechanisms. Bold for Delphi environment is the active event-organized system using its own isolated from operation system mechanism of messages-events exchange. The developer is provided with capabilities for forming his own kinds of events, and program realization of a subscription. Program generation of derived attributes and a unique possibility of using reverse-derived attributes are also based in many respects on subscribing mechanisms to events.

Chapter 14. Additional Capabilities Review

In this chapter additional tools and capabilities of Bold for Delphi program system, which were not considered in the previous chapters, are presented as the brief review. If necessary to get more detailed information it is possible to use the documentation and the demonstration examples included in product delivery.

Regions

Regions are used for blocking realization at a level of model elements at multi-user work. In practice of databases applications development situations occur, when it is necessary to provide protection of some data set from simultaneous change by different users. We shall consider, for example, a situation when in model there are time points of any process beginning and completion. Naturally, at that time of the process completion should exceed time of the beginning. But at multi-user work situations can take place, when one user edits the moment of the beginning, and another user – independently of the first one – edits the end point in such a manner that the logic of the data will be broken, and in result time of the process ending becomes less than time of the beginning. For protection from such situations both moments of time are necessary to be placed in one region. At that the following rule acts: the elements belonging to one region cannot be changed simultaneously by several users. Thus, when using regions, contents of region are protected (blocked) from actions of other users until the first user stop a session with elements of the given region. The element can be included also in several regions. In this case the following rule acts: for the element protection from change, all regions included the given element are blocked.

For regions realization Bold for Delphi provides support of special language of regions definition – RDL (Region Definition Language). RDL capabilities provide creation of regions, classes and attributes inclusion in region structure, and also forming *subregions* (thus, several subregions can be included into one region). Use of the subregions is convenient for including in them a class linked by association with the basic class. In this case protection of association and its roles values will be automatically provided. Bold for Delphi creates by default regions for model classes at "GenerateDefaultRegions" model tagged value set in "True". Regions created at that are formed by default according to the following rules:

8. Each class forms the region with "Default" name, into which the class itself is included, all single-multiple (a role multiplicity is no more than 1) links with other classes, and also all aggregate multiassociations (a role multiplicity is more than 1).
9. Each class forms the region with the "Exists" name, into which class members (attributes or associations) are not included. The given region provides protection from deleting objects of the given class.
10. For each aggregate link with other class the basic region with "Default" name includes similar region of an aggregated class as subregion.

11. For each non-aggregated single-multiple link one more region with "Default" name, including only the association itself is formed, at that the similar region of the linked class is included into this region as subregion.

The regions specification and forming their descriptions in RDL are carried out by input of test RDL-declarations in RegionDefinitions tagged value. This tagged value is determined for model on the whole. We shall remind (see Chapter 4), that access to an individual set of tagged values for the concrete model element is carried out by the following menu items of built-in editor: Tools ► Edit tagged values. Adjustment of tagged values also can be realized in Rational Rose (see Chapter 4).

Life Cycle of Linked Objects

Bold for Delphi allows setting at the model level a rule for controlling life cycle of objects existence. For this purpose, association role properties with "Delete action" and "Aggregation" names are used. These properties define, what will occur to associated ("subordinated") object, when deleting the main object. Rules of the specified properties implementation are the following (see Table 14.1).

Table 14.1.

#	«Delete action» property value	«Aggregation» property value	Description
1	Allow	Any	It is allowed to delete the main object without deleting the "subordinated" objects (integrity violation)
2	Prohibit	Any	It is forbidden to delete the min object (the program exception will be produced)
3	Cascade	Any	All linked objects are deleted automatically, when deleting the main one
3	<Default>	None	As in point #1
4	<Default>	Aggregate	As in point #2
5	<Default>	Composite	As in point #3

Three-tier Architecture

Bold for Delphi allows realizing applications with multi-tier architecture. In this case each architecture "tier" can represent a corresponding layer of program system – GUI, business-layer and persistence layer. In practice more often last two layers are located in one computer, while the GUI is taken out on client workplaces, forming so-called «thin clients». For organizing such applications in Bold for Delphi components structure there are special TBoldComConnectionHandle and TBoldSystemandleCom components. The last one is "representative" of a system handle on the client side, receiving all necessary information from a business-layer through DCOM-connection. The specific feature of creating multi-tier DCOM-applications is use of special visual components for forming the graphic interface and also a special set of the client object space handles.

At forming the client graphic interface of such applications there are no some convenient capabilities, for example, use of the built-in OCL-editor for generation of navigating OCL-

expressions is limited, and such expressions should be formed manually. It is possible to compensate these inconveniences partly if to use BoldClientifier utility included in product released version. This utility provides automatic transformation of the application for functioning as the thin client, at a level of updating the application project sources. The given utility can be found in the following folder: “..\BoldSoft\BFDR40D7Arch\Tools\BoldClientifier”.

The DB Remote Connection through SOAP

In considered Bold for Delphi version there is a capability of creating the remote persistence. It means a physical dislocation of a business-layer together with the GUI on one computer, and the database together with Persistence Layer providing mechanisms - on the remote computer. In this case SOAP (Simple Object Access Protocol) is applied for interoperability of these system components; it uses HTTP usual protocol as "transport". At that the remote persistence layer functions as usual WEB-server, and the DB data access can be achieved through Intranet-networks and through Internet global network. TBoldHTTPClientPersistenceHandle and TBoldWebConnection special components are applied for organizing the described scheme on the client side. On the server side TBoldHTTPServerPersistenceHandlePassthrough special component is used. It is necessary to say that programming of a server part of the considered scheme requires great efforts. However this will not stand in the way of realization and some companies (see Chapter 15) use the given mechanism successfully in the software products.

Object Space Synchronization

When working in the multi-user system own object space is formed in memory of each client. At that changes brought by each user in the object space (OS) by default don't have any effect on other users OS. Such situation can be the reason of occurrence of many difficulties and inconveniences at operation of information systems. In Bold for Delphi structure there is a special expansion of environment – the so-called propagator – the tool for OS synchronization (OSS-Object Space Synhchronization). Propagator is installed on the selected computer and functions as “a server of synchronization”, "listening" to each computer OS status, "subscribing" on the OS changes, and carrying out forced distribution of special notifications in client OS for their synchronization. For such capabilities involvement Bold for Delphi contains a set of special components, which can be found on <Bold OSS/CMS> bookmark of Delphi components palette.

Concurrency Management Server

The above described mechanism of the OS synchronization is inseparably linked with the blocking management mechanism or CMS-Concurrency Management Server. It is based on use of regions (see the beginning of this Chapter), and provides realization of "pessimistic" blocking at the level of set of simultaneously functioning object spaces. Blocking are carried out automatically at attempt to change the OS element value, at that both user

changes and element call from a program code are traced. For using concurrency management server, special components are intended; they are available on the same bookmark: <Bold OSS/CMS>.

Object Lending Library (OLLE)

OLLE library intended for replicating the data, contained in the several DB created from Bold for Delphi environment is one more expansion of Bold for Delphi program system. At that the matter concerns not so much direct copying objects, as more likely lending rights of access to the "own" object from "foreign" database. Such lent object can be used by a database only in reading mode, without possibility to change its value. Thus, in spite of the fact that the object is transferred at the disposal of another DB, "owner" of this object capable to change it is its "native" database. When changing the given object the notice on this change will be automatically transferred to those databases, to which the given object has been lent. For realization of such functionality components, which are located on <BoldOLLE> bookmark, are used.

Model and DB Evolution

In practice situations often occur, when it is necessary to make changes to UML-model (refinement of problem, correction of modeling errors, creation of the application new expanded versions, etc.). The model change in itself is realized in a simple way, but however, as it often happens, by the moment of such necessity occurrence the "old" application has already operated during some time interval, and, accordingly, a database, with which this application worked, has already been filled with the certain information. And at simple generation of new database structure according to the improved model, all information entered earlier will be irretrievably lost. Bold developers have made attempt of realizing special mechanisms, which use will help in many cases to solve such problems without serious consequences. These mechanisms are named Model Evolution and Database Evolution. Use of the model evolution (development or change) mechanism is based on the following basic approach: different model versions can function with the same database. This approach is realizable in many respects due to Bold for Delphi capabilities to generate automatically database schemes on UML-model. In the model evolution mechanism special means are included; they are intended for transformation of old model and database into their variants, with which both the new application and the "old" one can work. For controlling the process of model and database transformation the set of special tagged values is used. For example, FormerNames tagged value is capable to store several versions of names for model concrete element (class, association, etc.). EvolutionState tagged value controls use of elements in different versions. So, for example, when setting this parameter at some class in "Removed" value, Bold for Delphi environment will "ignore" existence of this class as though it is absent in model, but at that generation in a DB of the table for this class is not cancelled, that allows to use such DB by old applications. Use of ModelVersion tagged value, when UseModelVersion tagged value is set in "true", will lead to automatic generating a new column in the DB table; in this column the number of the model version will be automatically put in correspondence to each element. Actually database evolution after creating a new model and its import into

Bold environment, is realized from the built-in models editor by means of setting the following menu items: Tools ► Evolve Database. In result regeneration process of database structure with automatic checking capability of saving the information in a DB will begin. At that the detailed protocol of all changes brought into the database will be presented to the developer. The described mechanisms are certainly rather useful for practical development.

Multilanguage Support

Bold for Delphi has the advanced capabilities of forming the user multilingual interfaces. For presenting concrete languages TBALanguage class containing the language description is intended. At that each instance of the given class represents separate concrete language. Each moment the system works with the only one (current) language determined by CurrentLanguage variable value. BoldSetCurrentLanguageByName method allows setting any available language as current language. TBAMLString type is intended for interpretation of model string elements in various languages. It allows "to subscribe" (see Chapter 13) on changes of the current language in order to automate change of the user interface when changing the current language. TBoldAsMLStringRenderer component-renderer (see Chapter 9) also provides automation of mapping multilingual string elements.

Debugging Tools

Debugging of the MDA-applications created in Bold for Delphi environment, has the certain specificity. First, Delphi integrated debugger is not capable to realize debugging of OCL expressions (we shall remind that such expressions can be formed and made active directly from the application program code). Second, using OCL2SQL mechanism (see Chapter 10) results in automatic generation of SQL-queries set. And, third, in Bold environment special events are generated (see Chapter 13). For applications debugging Bold developers provided a special instance of all program dcu-modules; it is located in ..\SLIB folder. By default in process of development program modules from ..\LIB folder are involved. For receiving a capability of the expanded debugging the developer should change in Delphi settings a way to the program library used at configuration, having specified SLIB folder instead of LIB folder. Besides it is necessary in project properties the to add BOLD_SLIB symbol (through Delphi menu items: Delphi ► Project ► Options ► Directories/Conditionals ► | Conditional Defines). After that it is necessary to recompile the project. In result the developer gets access to the following tools:

- ☐ Debugger of execution time (TboldSystemDebuggerFrm class) ;
- ☐ OCL-expressions tracer;
- ☐ SQL-queries tracer;
- ☐ Tracer of DB references (higher-level tool, as compared to SQL tracer, allowing tracing the concrete data samples) .

Summary

The presented list of the additional capabilities provided to the developer by Bold for Delphi environment and its extensions, allows expanding essentially possible variants of practical using the given product. And use of the advanced debugging tools provides quality and efficiency improvement of applications development.

Chapter 15. Products of Third-party Companies

Appearance of Bold for Delphi software product (we shall remind that its first version has appeared in 1998), possessing a set of qualitatively new and even unique capabilities, could not pass unnoticed for companies producing software. By now some companies have already released several versions of the software products created specially for development in Bold for Delphi program environment. In this chapter the brief review of third-party companies' products is presented.

BoldExpress Studio

Manufacturer: Neosight Technologies Limited (<http://www.neosight.com>).

Type of license: a commercial product. A trial version exists.

BoldExpress Studio is rather interesting and multipurpose software product. In fact its structure includes several independent products:

- ❑ BoldExpress XMLSerialisation -Model Driven XML Applications;
- ❑ BoldExpress SOAP - Model Driven Web Services;
- ❑ BoldExpress Security;
- ❑ BoldExpress Server;
- ❑ BoldRetina.

BoldExpress XMLSerialisation provides translation of the information contained in UML-model into description in XML language. Besides the given product contains tools for use of OCL2XML language, which allows to realize "XML-queries". Such queries are rather convenient for use in Intranet and Internet.

BoldExpress SOAP gives convenient capabilities for using XML web-services. It can be used for automatic transformation of usual Bold-applications into the web-services set.

BoldExpress Security provides protection, assignment of users' access rights, and information encryption.

BoldExpressServer realizes the functionality similar to the usual Web-server, for management and distribution of the created applications in networks - Internet.

BoldRetina allows by means of simple way to replace completely Bold graphic level, having transformed it into the Web-interface; at that almost all capabilities on editing, forming windows, menu, etc. are kept.

Summarizing, it is possible to claim that BoldExpress is very successful extension and addition of Bold for use in corporate networks and Internet Global Network. On the website of company-developer it is possible to work "on-line" with WEB-application directly through the browser interface (see Fig. 15.1).

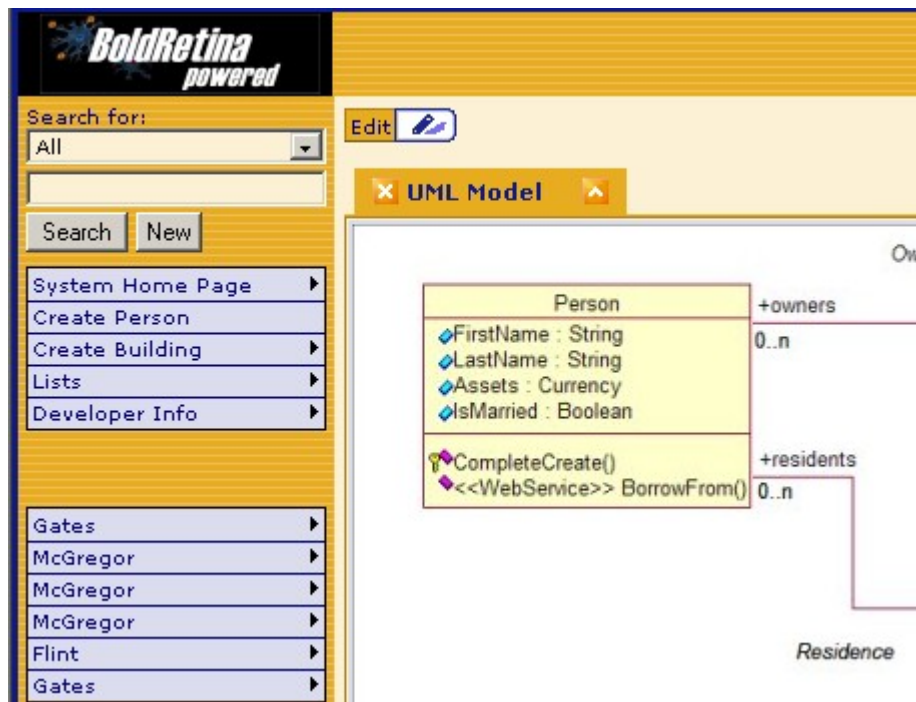


Fig. 15.1. A fragment of the demo online-application

At that it is possible to add and delete objects, to change attributes values, etc.

BoldGridPro

Manufacturer: Vipper Software (<http://www.rad-studio.com>).

Type of license: a commercial product. A trial version exists.

BoldGridPro component represents a grid for mapping the data with advanced interface capabilities (see Fig. 15.2).

Drag a column header here to group by that column					
Vendors					
Debit sum ▼	Credit sum ▼	Balance ▼	Industry ▲▼	Date of last contract	
0,00p.	0,00p.				
4 000,00p.	3 180,00p.	820,00p.	Computer Hardware	24 июн, 1999	
14 000,00p.	12 700,00p.	1 300,00p.	Telecommunications	03 янв, 1986	
4 500,00p.	8 300,00p.	-3 800,00p.	Computer Hardware	25 окт, 2002	
8 500,00p.	8 900,00p.	-400,00p.	Computer Hardware	12 июн, 2002	
40 000,00p.	37 000,00p.	3 000,00p.	Medical Services	24 янв, 2001	
46 000,00p.	47 000,00p.	-1 000,00p.	Aerospace/Defense	04 июн, 1999	
80 500,00p.	81 000,00p.	-500,00p.	Auto & Truck	02 июн, 2002	
6 300,00p.	7 190,00p.	-890,00p.	Telecommunications	09 ноя, 2000	
1 213,00p.	2 213,00p.	-1 000,00p.	Medical Services	09 июн, 1989	
70 090,00p.	78 900,00p.	-8 810,00p.	Hotel/Gaming	17 июн, 2002	
2 000,00p.	1 980,00p.	20,00p.	Telecommunications	24 фев, 1998	
140,00p.	889,00p.	-749,00p.	Aerospace/Defense	16 янв, 2002	
1 780,00p.	5 770,00p.	-3 990,00p.	Computer Hardware	06 май, 2001	
11 000,00p.	14 000,00p.	-3 000,00p.	Medical Services	12 янв, 2001	
1 000,00p.	576,00p.	424,00p.	Auto & Truck	25 июл, 2002	
5 100,00p.	5 100,00p.		Computer Software	09 окт, 1999	
31 000,00p.	32 000,00p.	-1 000,00p.	Medical Services	03 апр, 1994	
MAX=90 000,00p.				MAX=25 окт, 2002	

Fig. 15.2. BoldGridPro grid

BoldGridPro provides the following capabilities:

- ❑ The multiline headings formed both at the application development, and its run;
- ❑ Records grouping, sorting and filtering (see Fig. 15.3);
- ❑ Settings storing during the application run;
- ❑ Convenient adjusted panels - footers, for mapping summary automatically calculated values;
- ❑ Automatic incremental record search;
- ❑ Multichoice of records range;
- ❑ Individual adjustment of color and font for each cell;
- ❑ Export into PDF, Excel, HNML formats.

Industry ▲▼					
Vendors					
	Name ▲▼	Debit sum ▼	Credit sum ▼	Balance ▼	Date of
+ Industry:			(All)		
+ Industry: Aerospace/Defense			(Custom...)		
- Industry: Auto & Truck			0,00p.		
			576,00p.		
<input checked="" type="checkbox"/>	Dive & Surf	80 500,00p.	889,00p.	-500,00p.	02 июн
<input type="checkbox"/>	Marine Camera & Dive	1 000,00p.	1 520,00p.	424,00p.	25 июл
<input type="checkbox"/>	Techniques	1 257,00p.	1 980,00p.	-743,00p.	23 дек,
			2 000,00p.		
- Industry: Computer Hardware			2 213,00p.		
<input type="checkbox"/>	Amor Aqua	4 000,00p.	3 180,00p.	820,00p.	24 июн
<input type="checkbox"/>	B&K Undersea Photo	4 500,00p.	5 100,00p.	-3 800,00p.	25 окт,
<input type="checkbox"/>	Beauchat, Inc.	8 500,00p.	5 770,00p.	-400,00p.	12 июн
<input checked="" type="checkbox"/>	Glen Specialties, Inc.	1 780,00p.	7 190,00p.	-3 990,00p.	06 май,
<input type="checkbox"/>	Perry Scuba	10 000,00p.	8 300,00p.	1 080,00p.	07 окт,
			8 920,00p.		
+ Industry: Computer Software					
+ Industry: Hotel/Gaming					
+ Industry: Medical Services					
+ Industry: Telecommunications					
MAX=90 000,00p.				MAX=2	

Fig. 15.3. Grouping and filtering

Besides BoldGridPro, the company offers also some other visual components for realizing the GUI. You can find the information on them on the developer web-site.

OCL Extensions

Manufacturer: Holton Integration Systems (<http://www.holtonsystems.com>).

Type of license: a shareware product with sources.

This is a package of components for extension of OCL operators structure. The set of additional operators includes about 80 new kinds of expressions which can be used for forming OCL expressions. After the package installation all new OCL-expressions become accessible to use both during the program run, and at the developing stage, in Bold for Delphi built-in OCL-editor. Some of the provided additional capabilities are described below (see Table 15.1).

Table 15.1. Some additional OCL operators

OCL-expression	Description
ThisBoldOCLBoldID	Returns BoldID object identifier
TThisBoldOCL.Sqrt	Square root calculation
TThisBoldOCLasInteger	Transforms the logic value into the integer (false=0, true=1)
TThisBoldOCLasCommaList	Transforms collection into the list of values separated by «;». It is convenient for export into StringList

Most of OCL-extensions are intended for work with dates and time. Taking into account that it is free product, it can be recommended for downloading from the developer web-site.

Bold TCP OSS

Manufacturer: Holton Integration Systems (<http://www.holtonsystems.com>).

Type of license: commercial distribution. A trial version exists.

The product is intended for realizing Object Space Synhronization mechanism – synchronization of object spaces (see Chapter 14) by means of the TCP-protocol.

Bold SOAP Server (BSS)

Manufacturer: sourceforge (<http://sourceforge.net/projects/boldsoapserver/default.htm>)

Type of license: a shareware product with sources.

BSS is a software shell (wrapper) round BoldSystem providing WEB-access to Bold-application (see also BoldExpress Studio in the beginning of the current Chapter).

BoldRave

Manufacturer: Intrasting (<http://intrasting.com/boldrave/>)

Type of license: a shareware product with sources.

The package of components provides direct generation of Nevrona Rave reports from Bold-applications, without necessity of using program adapters of TboldDataSet type (see Chapter 11). It supports handles of OS variables, possesses rich capabilities for adjusting the report kind and contents, also during the application run, and provides preview before printing (see Fig. 15.4).

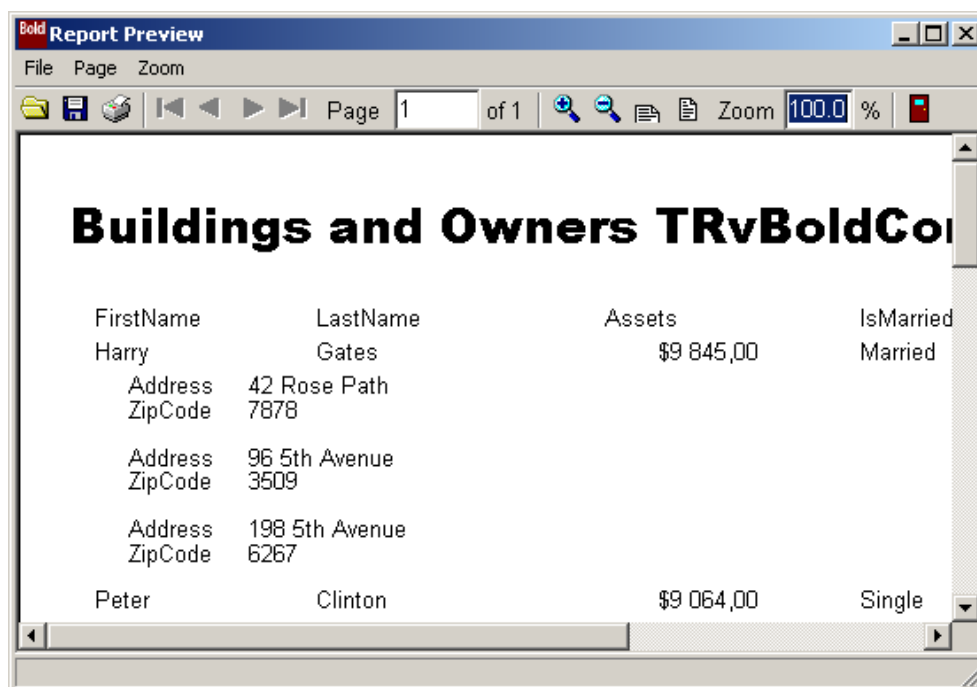


Fig. 15.4. Report preview window (BoldRave)

deBold

Manufacturer: DroopyEyes (<http://www.droopyeyes.com>)

Type of license: a shareware product with sources.

The package consists of several useful components intended for checking a correctness of state of OS objects and handles, constraints for objects and forms of the application as a whole. It also includes rather interesting components: TdeBoldAutoIncManager and TdeBoldAutoIncProvider for realization of the user autoincremental attributes.

phGanttTimePackage и phGrid_BA

Manufacturer: PlexityHide (<http://www.plexityhide.com>).

Type of license: commercial distribution. A trial version exists.

phGanttTimePackage is the package of components for the automated constructing diagrams, time scales, schedules and charts. It possesses the advanced capabilities of the interface adjustment (see Fig. 15.5)



Fig. 15.5. Using phGanttTime

phGrid_BA is grid with the advanced capabilities of mapping the data (see Fig. 15.6). For example, it allows including the enclosed grids into the basic grid cells.

Form1			
Persons and buildings		Personal data	
<div> <div>Persons</div> <div> <div>Peter Flint</div> <div>8952</div> <div>Married</div> <div><input checked="" type="checkbox"/></div> </div> <div>John Gates</div> <div>10544</div> <div>Married</div> <div><input type="checkbox"/></div> </div> <div> <div>Harry Flint</div> <div>8833</div> <div>Married</div> <div><input type="checkbox"/></div> </div>			

Fig. 15.6. phGrid_BA grid

Summary

The conclusion can be made from the presented brief review: there are many third-party program extensions and additions for Bold for Delphi; they realize rather various and useful additional capabilities for creation of applications in Borland MDA environment.