Eiffel for .NET Binding
for db4o

User's and Developer's Manual

Ruihua Jin, Marco Piccioni

ETH Zurich, Chair of Software Engineering

ETH Zentrum, RZ Building

CH-8092 Zurich, Switzerland

rjin@student.ethz.ch, marco.piccioni@inf.ethz.ch

April 3, 2008

Table of Contents

1 Introduction

1.1 Summary

Db4o is an already established OODBMS solution for Java and .NET, providing a powerful and easy-to-use solution for object persistence.

It is therefore desirable to make it accessible to programmers that use Eiffel, a well-known, pure object-oriented programming language offering features like Design by Contract, multiple inheritance, genericity and agents. Though Eiffel participated to the birth of the .NET Framework and was integrated in it from the very start, it is not trivial that db4o can flawlessly persist Eiffel objects as well as, say, C# objects. The aim of this project is to identify peculiarities of persisting Eiffel objects and to provide solutions so that Eiffel developers can use db4o as seamlessly as possible.

The effort that this documentation describes is the implementation of the necessary db4o framework classes to make it usable within Eiffel applications.

1.2 Structure of This Document

Chapter 2 gives an overview of how Eiffel types are mapped to .NET types without loosing the multiple inheritance hierarchy information. Chapter 3 discusses the three db4o querying mechanisms: Query-By-Example, SODA Query API and Native Queries, showing how we can adapt them for querying Eiffel objects. Then chapter 4 shows how to configure the activation depth, update depth, delete behavior and indexing. Chapter 5 discusses refactoring of Eiffel classes. Chapter 6 introduces the classes implemented for the current project. Chapter 7 illustrates how to use .NET delegates in Eiffel for db4o callbacks. Chapter 8 shows how to use C# to persist Eiffel objects and the other way around, and chapter 9 shows how to use Eiffel to persist C structs. At the end we draw some conclusions.

2 Mapping of Eiffel Types to .NET Types

Eiffel and the .NET Framework have two different type systems. It is therefore important to first explore the mapping strategy of Eiffel types to .NET types before getting on with persistence of Eiffel objects in db4o databases.

2.1 Mapping of Eiffel Built-in Expanded Types to .NET CTS Types

The Eiffel expanded types defined in EiffelBase library (such as `BOOLEAN` and `INTEGER`) are directly mapped to the types of the .NET Common Type System (CTS). Table 1 shows the mapping.

Table 1. Mapping of Eiffel built-in expanded types to .NET CTS types

Eiffel basic type	Equivalent CTS type
`BOOLEAN`	`System.Boolean`
`CHARACTER_8`	`System.Char`
`CHARACTER_32`	`System.UInt32`
`INTEGER_8`	`System.SByte`
`INTEGER_16`	`System.Int16`
`INTEGER_32`	`System.Int32`
`INTEGER_64`	`System.Int64`
`NATURAL_8`	`System.Byte`
`NATURAL_16`	`System.UInt16`
`NATURAL_32`	`System.UInt32`
`NATURAL_64`	`System.UInt64`
`REAL_32`	`System.Single`
`REAL_64`	`System.Double`

The mapping is quite straightforward. However, notice that type `CHARACTER_32` becomes of type `System.UInt32` at run-time. So if the developer has a class which contains a field of type `CHARACTER_32`, then the field would be stored as `System.UInt32` in a db4o database.

2.2 Mapping of Eiffel Expanded Types to .NET Value Types

Besides the built-in expanded types described above, Eiffel developers can define their own expanded types. One simple example is an expanded `POINT` class which has two attributes for the x- and the y-coordinates.

```
expanded class
   POINT

create
   default_create,
   make_with_x_y

feature
   make_with_x_y(a_x: INTEGER; a_y: INTEGER) is
      do
         set_x(a_x)
         set_y(a_y)
      end

   set_x(a_x: INTEGER) is
      do
         x := a_x
      end

   set_y(a_y: INTEGER) is
      do
         y := a_y
      end

   x, y: INTEGER

end
```

The Eiffel for .NET compiler generates five .NET types for `POINT`, they are

```
Point
ReferencePoint
Impl.ReferencePoint
Create.Point
Create.ReferencePoint
```

The UML diagram in Fig. 1 shows their inter-type relationships along with the fields and methods each type contains (for the sake of clarity, fields and methods in `ValueType`, `EIFFEL_TYPE_INFO`, Any are omitted).

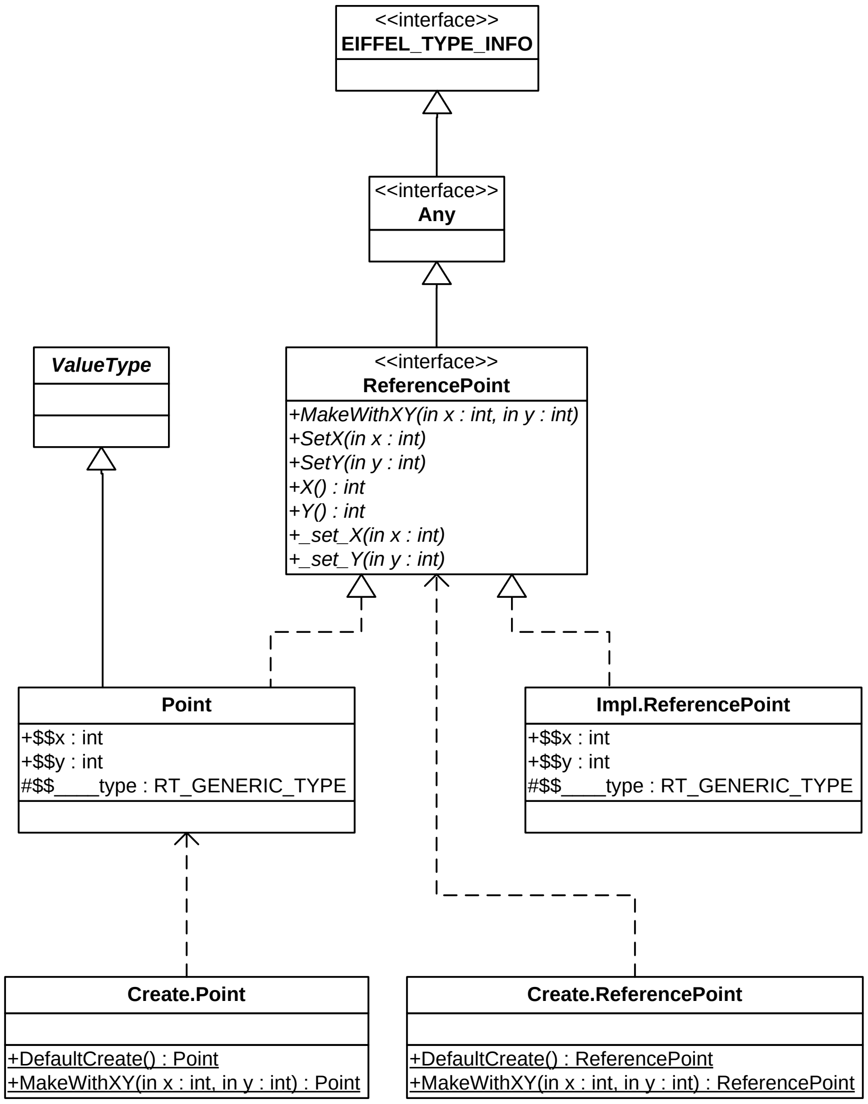


Fig. 1. The .NET types generated for the Eiffel class `POINT` and the inter-type relationships between them.

Interface `EIFFEL_TYPE_INFO`, defined in `eiffelsoftware.runtime.dll`, is the root interface implemented by all Eiffel for .NET types, and interface `Any` is the interface counterpart for the Eiffel class `ANY` (the implementation class of interface `Any` is `EiffelSoftware.Library.Base.Kernel.Impl.Any` which is defined in `EiffelSoftware.Library.Base.dll`).

Interface `ReferencePoint` contains methods which are defined in class `POINT`. `MakeWithXY`, `SetX` and `SetY` correspond to `make_with_x_y`, `set_x` and `set_y` (note that the routine names are adapted to the .NET naming conventions, and so are the attribute names, see below). Methods `X` and `_set_X` are the getter and setter for the attribute `x`, methods `Y` and `_set_Y` are the getter and setter for the attribute `y`.

Classes `Point` and `Impl.ReferencePoint` both implement interface `ReferencePoint`, and class `Point` inherits from `ValueType`. Fields `$$x` and `$$y` are the .NET counterparts of the attributes `x` and `y`, with their names adapted to the .NET naming conventions. Field `$$____type` is used to store type information of actual generic parameters in case of a generic type.

The two classes in namespace `Create` are, as the name suggests, for creation and initialization of the related objects.

Note that in Eiffel for .NET, expanded types cannot inherit from any other expanded types, which is one of the few restrictions brought about by mapping Eiffel to .NET.

2.3 Mapping of Eiffel Reference Types to .NET Types

Since the Common Language Runtime of the .NET Framework only supports single inheritance of classes, the primary concern with the Eiffel integration was how to preserve the multiple inheritance structure of Eiffel types. This issue was solved using the "simulated" multiple inheritance structure of .NET interfaces.

For example, we have four Eiffel classes `PARALLELOGRAM`, `RHOMBUS`, `RECTANGLE` and `SQUARE`, with `RHOMBUS` and `RECTANGLE` inheriting from `PARALLELOGRAM`, and `SQUARE` inheriting from both `RHOMBUS` and `RECTANGLE` (see Fig. 2). Furthermore, `RECTANGLE` renames `height1` as `width`, `height2` as `height`, and `SQUARE` renames `width` as `side_length`. As we will see later, feature renaming requires special attentions in using SODA Query API.

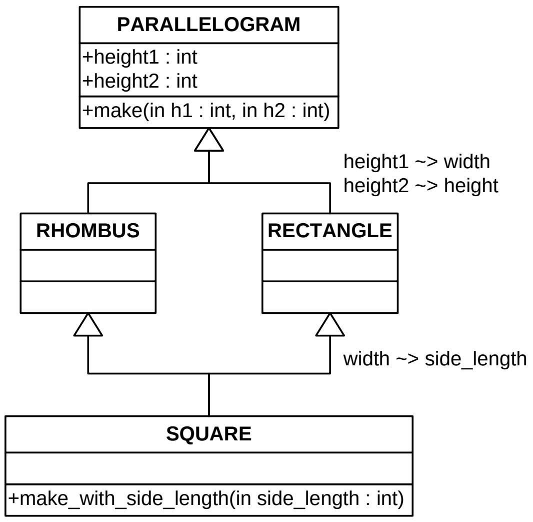


Fig. 2. Four Eiffel classes with their inter-class relationships.

When an effective (fully implemented) Eiffel class, say `SQUARE`, is compiled for the .NET Framework, three .NET types will be generated: the interface `Square`, the class `Impl.Square` which inherits from `System.Object` and implements interface `Square`, and the class `Create.Square` whose static method is generated according to the creation procedure in `SQUARE` and is therefore used to create and initialize instances of `Impl.Square`. The inheritance of `SQUARE` from `RHOMBUS` and `RECTANGLE` is preserved through the inheritance of the `Square` interface from the `Rhombus` and `Rectangle` interfaces. Fig. 3 shows the whole picture of the .NET counterpart (for the sake of clarity, `Create.X` classes are omitted in the figure).

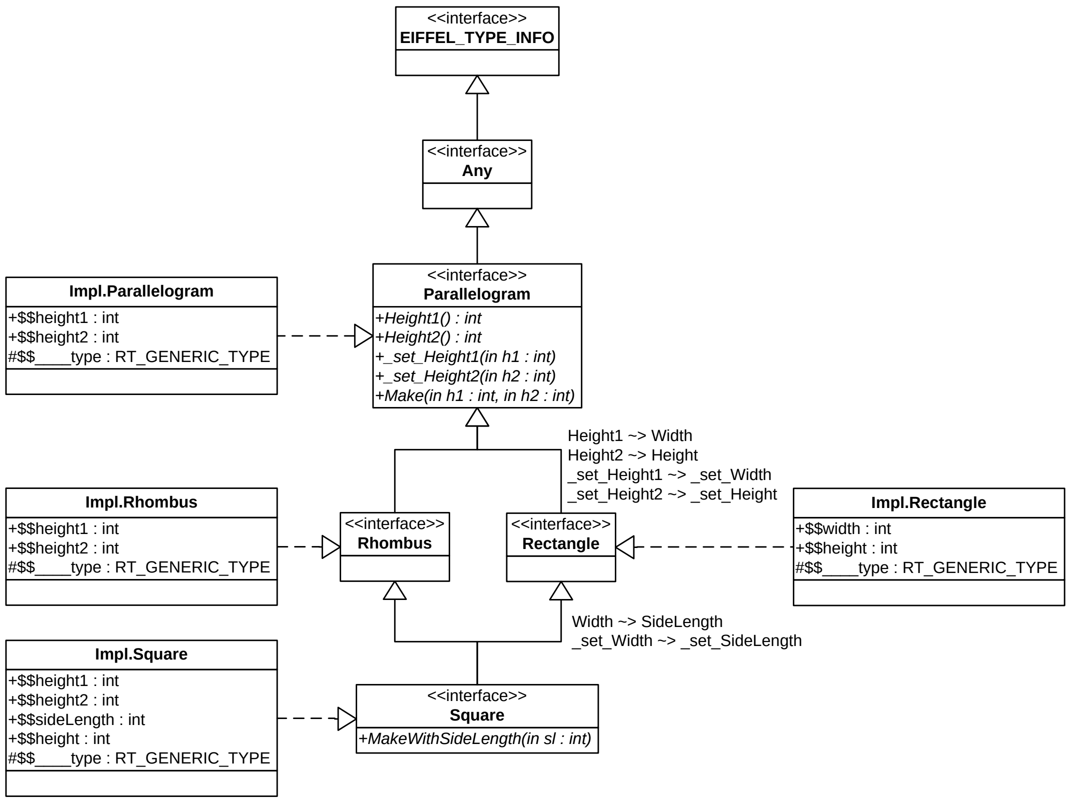


Fig. 3. The corresponding .NET types with their inter-type relationships.

For the sake of completeness, a deferred (abstract) Eiffel class, say `X`, is compiled to two .NET types: the interface `X` and the abstract class `Impl.X`. `Create.X` is not generated for obvious reasons.

As a matter of fact, in .NET all implementation classes `Impl.X` are direct subclasses of `System.Object`. They are coupled with each other only through interfaces. As a result of this special mapping strategy, when querying for Eiffel objects we should always use interfaces as query extents to get correct results.

2.4 Feature Adaptations in Inheritance

Eiffel brings to developers not only multiple inheritance, but also feature adaptation techniques along with it. They are feature renaming, change of export status, undefinition, redefinition and selection. Now we are going to look into the .NET types to see how feature adaptations are realized in there.

2.4.1 Renaming an Attribute

In the example described in the previous section, `RECTANGLE` renames the inherited attributes `height1` as `width` and `height2` as `height`. This change affects the generation of `Rectangle` and `Impl.Rectangle`:

```
interface Parallelogram : Any
{
   int Height1();
   int Height2();
   void _set_Height1(int h1);
   void _set_Height2(int h2);
   ...
}

interface Rectangle : Parallelogram
{
   int Width();
   int Height();
   void _set_Width(int w);
   void _set_Height(int h);
   ...
}

class Impl.Rectangle : Rectangle
{
   public int $$width;
   public int $$height;
   public int Width() { ... }
   public int Height() { ... }
   public void _set_Width(int w) { ... }
   public void _set_Height(int h) { ... }
   ...
}
```

Basically in .NET every time you do a renaming, the interface for the class doing the renaming adds a new method for that renamed feature, and in the implementation class the new method is declared to implement the old method declaration. The declaration is done through the attribute `MethodImplAttribute`.

In our example with two attribute renaming, four new methods are added in interface `Rectangle`. Since the four “old” methods declared in interface `Parallelogram` are not part of the class interface of `Impl.Rectangle` anymore, the four new ones are declared to implement the four old ones. The MSIL disassembler `ildasm.exe` reveals the code in the `Impl.Rectangle.Width()` method and we can see that `Width` implements `Height1`:

```
.method public hidebysig virtual instance int32
       Width() cil managed
{
   .override RootCluster.Parallelogram::Height1
   // Code size       7 (0x7)
   .maxstack  8
   IL_0000:  ldarg.0
   IL_0001:  ldfld  int32 RootCluster.Impl.Rectangle::$$width
   IL_0006:  ret
} // end of method Rectangle::Width
```

Corresponding to attributes `width` and `height` in `RECTANGLE`, the two fields in `Impl.Rectangle` are named `$$width` and `$$height`.

2.4.2 Renaming a Routine

Suppose `RECTANGLE` renames the routine `make` as `make_with_width_height`. We then have the following changes in `Rectangle` and `Impl.Rectangle`:

```
interface Parallelogram : Any
{
   void Make(int h1, int h2);
   ...
}

interface Rectangle : Parallelogram
{
   void MakeWithWidthHeight(int w, int h);
   ...
}
```

```
class Impl.Rectangle : Rectangle
{
   void MakeWithWidthHeight(int w, int h) { ... }
   ...
}
```

with `MakeWithWidthHeight` implementing `Make` in `Impl.Rectangle`.

2.4.3 Exporting Features

In the example application, we defined the `SQUARE` class as follows:

```
class
   SQUARE

inherit
   RECTANGLE
      rename
         width as side_length
      export
         {NONE} height
      select
         side_length, height
      end

   RHOMBUS
      export
         {NONE} height1, height2
      end

create
   make_with_side_length

feature
   make_with_side_length(sl: INTEGER) is
      require
         sl_positive: sl > 0
      do
         make(sl, sl)
         height1 := sl
         height2 := sl
      end

end
```

Although features `height`, `height1` and `height2` are exported to {`NONE`}, their corresponding fields

- `$$height,`
- `$$height1,`
- `$$height2`

and their corresponding getter and setter methods

- `Height, _set_Height,`
- `Height1, _set_Height1`
- `Height2, _set_Height2`

in class `Impl.Square` all have a visibility scope of `public`. However, this issue should not be a concern of the project, since db4o stores all the fields of an object, regardless whether they are public, protected or private.

2.4.4 Undefining a Routine

The corresponding method in the .NET implementation class is implemented according to the routine which is not undefined.

2.4.5 Redefining a Routine

The corresponding method in the .NET implementation class is implemented according to the new implementation of the routine in the Eiffel class.

2.4.6 Turning an Argumentless Function into an Attribute

Suppose we have two classes, `SYMMETRY` and `PARALLELOGRAM`:

```
deferred class
    SYMMETRY

feature
    center: POINT is
            -- Returns the center of the geometry figure.
        deferred
        end
end
```

Let’s further suppose that in class `PARALLELOGRAM`, the routine `center` is redefined as an attribute.

```
class
    PARALLELOGRAM

inherit
    SYMMETRY
        redefine
            center
        end

feature
    center: POINT

...
end
```

In the .NET counterpart we then have

```
interface Symmetry : Any
{
    Point Center();
}

interface Parallelogram : Symmetry
{
    Point Center();
    void _set_Center(Point p);
    ...
}
```

```
class Impl.Parallelogram : Parallelogram
{
    Point $$center;
    Point Center();
    void _set_Center(Point p);
    ...
}
```

As shown, two new methods are added in the interface `Parallelogram`, and one new field `$$center` is added in the class `Impl.Parallelogram`.

2.4.7 Selecting a Feature

Feature selection only plays a part in dynamic binding. Since in Eiffel for .NET, all the implementation classes do not have any subclasses, the only concern about calling the right method in an implementation class is reduced to the question of how to find out the new method which implements the old method declared in an interface, which is solved easily with the help of `MethodImplAttribute`.

2.5 Mapping of Eiffel Generic Types to .NET Types

An Eiffel generic type, say `GLIST[G]`

```
class
    GLIST[G]

feature
    item: G

end
```

is mapped to the following .NET types if the actual parameter `G` is of a reference type:

```
interface GlistReference : Any
{
   RT_TYPE _db4o_for_eiffel_type_9986();
   RT_TYPE _db4o_for_eiffel_type_9987();
   RT_TYPE _GLIST_Formal#1();
   void _set_Item(object);
   object Item();
}

class Impl.GlistReference : GlistReference
{
   RT_GENERIC_TYPE $$____type;
   object $$item;
   ...
}
```

As `GLIST[STRING]` and `GLIST[RECTANGLE]` would both become of type `GlistReference` at run-time, the two would conform to each other from the point of view of the .NET run-time system. The Eiffel for .NET run-time system, however, knows the type information of actual generic parameters. Taking advantage of this, we implemented a helper class called `GENERICITY_HELPER` which inherits from `INTERNAL` in assembly `eiffelsoftware.runtime.dll`, and its routines can be used to decide the conformance between two generic or non-generic objects according to the Eiffel's conformance rule.

Eiffel's conformance rule. A type `U` conforms to a type `T` only if the base class of `U` is a descendant of the base class of `T`; also, for generically derived types, every actual parameter of `U` must (recursively) conform to the corresponding formal parameter in `T`.

The developer should always make use of this helper class to filter out non-conforming objects when querying for a generic class of reference types. See chapter 3 for details.

If we have a constrained generic type like

```
class GLIST[G -> PARALLELOGRAM]
```

then in interface `GlistReference` we have

```
interface GlistReference : Any
{
   RT_TYPE _db4o_for_eiffel_type_9986();
   RT_TYPE _db4o_for_eiffel_type_9987();
   RT_TYPE _GLIST_Formal#1();
   void _set_Item(Parallelogram);
   Parallelogram Item();
}
```

and in class `Impl.GlistReference` we have

```
class Impl.GlistReference : GlistReference
{
   RT_GENERIC_TYPE $$____type;
   Parallelogram $$item;
   ...
}
```

With a constraint on the formal parameter `G`, we cannot create instances of `GLIST[X]` if `X` is not a descendant class of `PARALLELOGRAM`. So a query for `GLIST[PARALLELOGRAM]` or `GLIST[RECTANGLE]` objects would only return those objects which conform to `GLIST[PARALLELOGRAM]`. A constraint on formal parameters therefore helps to get a more fine-grained query result even if the conformance of objects in a query result is not always guaranteed.

On the other hand, if the actual parameter `G` is of an expanded type, say `POINT` as in a previous example, then two more .NET types would be generated:

```
interface GlistPoint : Any, GlistReference
{
   RT_TYPE _db4o_for_eiffel_type_9986();
   RT_TYPE _db4o_for_eiffel_type_9987();
   RT_TYPE _GLIST_Formal#1();
   void _set_Item(Point);
   Point Item();
}

class Impl.GlistPoint : Any, GlistPoint
{
   RT_GENERIC_TYPE $$____type;
   Point $$item;
   ...
}
```

As you see, for each generic class of expanded types, a new interface and a class which implements this interface will be generated. This approach ensures that GLIST[RECTANGLE] and GLIST[POINT] objects will become of different .NET types at run-time: GLIST[RECTANGLE] objects are of type GlistReference, and GLIST[POINT] objects are of type GlistPoint. This also means that the developer would get a correct query result when querying for GLIST[POINT] or GlistPoint objects.

2.6 Tuples

The .NET interface for tuples is

```
interface EiffelSoftware.Library.Base.Kernel.Dotnet.Tuple :
Hashable
```

and the corresponding implementation class is

```
class EiffelSoftware.Library.Base.Kernel.Dotnet.Impl.Tuple
    : Hashable, Tuple
{
    RT_GENERIC_TYPE $$____type;
    object[] $$nativeArray;
    ...
}
```

Both are defined in the assembly EiffelSoftware.Library.Base.dll.

All tuples are generic, and the actual generic parameters are stored in an array of type object. The developer also needs to use the provided helper class GENERICITY_HELPER to get conforming objects when querying for a certain kind of tuple.

3 Querying for Eiffel Objects in db4o Databases

Db4o supplies three querying systems: Query-By-Example, SODA Query API and Native Queries. We are now going to compare them from the point of view of Eiffel applications.

3.1 Query-By-Example

In Query-By-Example we provide the object container with a template object, and then the object container returns all the objects whose fields match all the non-default fields of the template object.

Since in .NET all instances of Eiffel types don't directly inherit from each other but are only related to each other through interfaces, Query-By-Example, which uses the .NET reflection mechanism to determine subclasses of a class, is only able to retrieve direct instances of the template type. For example, the query

```
retrieve_rectangle is
    local
        object_container: IOBJECT_CONTAINER
        template: RECTANGLE
        query_result: IOBJECT_SET
        closed: BOOLEAN
    do
        object_container := {DB_4O_FACTORY}.open_file("f.db4o")
        create template.make(10, 0)
        query_result := object_container.get(template)
        closed := object_container.close
    rescue
        if (object_container /= Void) then
            closed := object_container.close
        end
    end
```

only returns direct instances of `RECTANGLE` with `width=10`, while `SQUARE` objects are not returned. This is because `template_object` becomes of type `Impl.Rectangle` at run-time, and `Impl.Square` does not inherit from `Impl.Rectangle`.

Query-By-Example already has several limitations by its own, and only fits with very simple queries. Although a wrapper class for Query-By-Example which returns not only the direct instances but also instances of subclasses could be implemented, we did not implement it because it would not overcome the restricted functionalities that are part of the approach itself.

3.2 SODA Query API

The SODA Query API provides Eiffel applications with an efficient, though type-unsafe way of querying for objects. There are three issues we should be aware of when we are querying for Eiffel objects with SODA Query API, and they originate from

- feature name translation for the sake of the .NET naming conventions,
- feature renaming in descendant classes, and
- mapping of Eiffel generic types to .NET types.

3.2.1 Using Valid Field Names

When building a query graph, a SODA query “descends” to a field by specifying the field name. The following query specifies the interface `{RHOMBUS}` as the query extent and descends to field `$$height1` to constrain it with the value 10. The query retrieves all `RHOMBUS` (including `SQUARE`) objects whose `height1` is equal to 10 (`({RHOMBUS}).to_cil` is the Eiffel notation for the .NET interface type of `RHOMBUS`):

```
retrieve_rhombus1 is
    local
        object_container: IOBJECT_CONTAINER
        iquery: IQUERY
        iconstraint, isubconstr: ICONSTRAINT
        query_result: IOBJECT_SET
        closed: BOOLEAN
    do
        object_container := {DB_4O_FACTORY}.open_file(“f.db4o”)
        iquery := object_container.query
        iconstraint := iquery.constrain(({RHOMBUS}).to_cil)
        isubconstr := iquery.descend(“$$height1”).constrain(10)
        query_result := iquery.execute
        closed := object_container.close
```

```
      rescue
         if (object_container /= Void) then
            closed := object_container.close
         end
      end
```

As mentioned above, Eiffel for .NET compiler adapts attribute names to the .NET naming conventions and also prepends “$$” to each if the attribute is defined in an Eiffel class. The developer could use our wrapper classes `QUERY` and `CONSTRAINT` for the db4o `IQUERY` and `ICONSTRAINT` interfaces to have field name translation done by the wrapper. For a query like above, we could write

```
   retrieve_rhombus2 is
      local
         object_container: IOBJECT_CONTAINER
         query: QUERY
         constraint, sc: CONSTRAINT
         query_result: IOBJECT_SET
         closed: BOOLEAN
      do
         object_container := {DB_4O_FACTORY}.open_file(“f.db4o”)
         create query.make_from_query(object_container.query)
         constraint := query.constrain({RHOMBUS})
         sc := query.descend("height1", {RHOMBUS}).constrain(10)
         query_result := query.execute
         closed := object_container.close
      rescue
         if (object_container /= Void) then
            closed := object_container.close
         end
      end
```

3.2.2 Including Attribute Names Renamed in Descendant Classes

The second issue with SODA Query API originates from renaming a feature in descendant classes. In our example, if we rename the feature `width` to `side_length` in the `SQUARE` class, then a SODA query for `RECTANGLE` objects will not return `SQUARE` objects, because the `Impl.Square` class only has a field called `$$sideLength` instead of `$$width`. To retrieve a correct query result, we have to modify the query as follows:

```
iquery := object_container.query
iconstraint := iquery.constrain({RECTANGLE})
wcon := iquery.descend("$$width").constrain(10)
slcon := iquery.descend("$$sideLength").constrain(10)
isubconstr := wcon.or_(slcon)
query_result := iquery.execute
```

The two wrapper classes `QUERY` and `CONSTRAINT` also take this case into consideration, and they get the field value constraints OR-joined if the field is renamed in descendant classes. For the above query, we write

```
create query.make_from_query(object_container.query)
constraint := query.constrain({RECTANGLE})
sc := query.descend("width", {RECTANGLE}).constrain(10)
query_result := query.execute
```

The developer should be aware of the fact that having too many attribute renamings may cause performance loss in querying because of a larger query graph.

3.2.3 Querying for Generic Objects or Tuples

As described in chapter 2, when we are querying for generic objects of reference types, we may get non-conforming objects in the query result returned. In this case, we should use routine `get_conforming_objects` in class `GENERICITY_HELPER` to filter out all the non-conforming objects:

```
get_conforming_objects(a_list: ILIST;
            an_object: SYSTEM_OBJECT): LIST [SYSTEM_OBJECT]
         -- List of objects in `a_list'
         -- which conform to `an_object'
    require
        list_not_void: a_list /= Void
        an_object_not_void: an_object /= Void
```

For example, to get a correct query result for `GLIST[PARALLELOGRAM]`, we write

```
create query.make_from_query(object_container.query)
constraint := query.constrain({GLIST[PARALLELOGRAM]})
query_result := query.execute
result_list := get_conforming_objects(query_result,
                             create {GLIST[PARALLELOGRAM]})
```

Similar for tuples, to query for `TUPLE[GLIST[INTEGER]]`, we write

```
create query.make_from_query(object_container.query)
constraint := query.constrain({TUPLE[GLIST[INTEGER]]})
query_result := query.execute
result_list := get_conforming_objects(query_result,
                            create {TUPLE[GLIST[INTEGER]]})
```

3.2.4 SODA Query API for Eiffel Strings

Currently Eiffel developers cannot add constraints on an Eiffel string attribute in a SODA query. In fact the following call

```
iquery.descend("eiffel_str").constrain("a").starts_with(False)
```

would fail.

3.2.5 Using the Right Db4o Version

Db4o was not able to return correct query results of SODA queries until version 7.1.26, so the developer should always use a more recent version. See the related issue report on "SODA queries return wrong query results for .NET interfaces" at
http://tracker.db4o.com/browse/COR-1086 for details.

As of this writing, there is another unsolved bug related to SODA Query API, see "AND, NOT constraints return wrong query results" at http://tracker.db4o.com/browse/COR-1131 for details.

3.3 Native Queries

The most compelling plus of Native Queries is that they enforce a type-safe approach.

3.3.1 Native Queries for Eiffel Objects

The Eiffel developer creates a descendant class of `PREDICATE` class (since there is a name clash with `PREDICATE` in EiffelBase Library, the developer has to first rename one of them, in our example, we rename `PREDICATE` in assembly `Db4objects.Db4o.dll` as `DB4O_PREDICATE`) and defines a Boolean function `match` to run Native Queries. The `match` method tells the query engine whether to include or exclude a candidate object in the query result. Here is an example of querying for all `PARALLELOGRAM` objects whose `height1` is greater than 10:

```
class
    PARALLELOGRAM_PREDICATE

inherit
    DB4O_PREDICATE

feature
    match(p: PARALLELOGRAM): BOOLEAN is
            -- Is `p.height1' greater than 10?
        do
            Result := p.height1 > 10
        end

end
```

Then we pass an instance of `PARALLELOGRAM_PREDICATE` as argument to the `query` method of `IOBJECT_CONTAINER`:

```
retrieve_parallelogram is
    local
        object_container: IOBJECT_CONTAINER
        query_result: IOBJECT_SET
        closed: BOOLEAN
    do
        object_container := {DB_4O_FACTORY}.open_file("f.db4o")
        query_result := object_container.query(
                             create {PARALLELOGRAM_PREDICATE})
        closed := object_container.close
    rescue
        if (object_container /= Void) then
            closed := object_container.close
        end
    end
```

3.3.2 Native Queries for Generic Objects

If we want to query for generic objects of reference types, say `GLIST[PARALLELOGRAM]`, then we must make use of the `conforms_to_object` method in the helper class `GENERICITY_HELPER`:

```
conforms_to_object(obj1: ANY; obj2: ANY): BOOLEAN
         -- Does `obj1' conform to `obj2'?
    require
        obj1_not_void: obj1 /= Void
        obj2_not_void: obj2 /= Void
```

In the `match` method we first restrict the candidate objects to be of the same type as the template object `create {GLIST[PARALLELOGRAM]}`, then we define our own matching criteria:

```
class
    GLIST_PREDICATE

inherit
    DB4O_PREDICATE
    GENERICITY_HELPER

feature
    match(gl: GLIST[PARALLELOGRAM]): BOOLEAN is
            -- Does `gl' fulfill matching criteria?
        do
            Result := conforms_to_object(gl,
                                create {GLIST[PARALLELOGRAM]})
            Result := Result and then gl.item.height1 > 10
        end

end
```

For a query for generic objects of expanded types like `GLIST[POINT]`, we don't need to use the helper class.

3.3.3 Performance

The db4o team is making a big effort to optimize Native Queries so that they can be run against indexes. Eiffel applications, however, cannot take advantage of the optimization algorithms yet, which means, Native Queries in Eiffel applications cannot be optimized.

The reason is the following: the `match` method in subclasses of `DB4O_PREDICATE` is declared like

```
match(candidate: SOME_TYPE): BOOLEAN
```

`candidate` becomes of an interface type `SomeType` at run-time, and db4o cannot optimize interface method calls in the `match` method body.

The open question is therefore whether and how Native Queries in Eiffel applications can be optimized to be run against indexes.

3.3.4 Using Agents for Native Queries

In the .NET version of db4o the developer can use delegates for Native Queries. As Eiffel has its own powerful mechanism of modeling operations, called agents, we decided to integrate agents into the concept of Native Queries for Eiffel applications.

An agent is an encapsulation of a routine. A typical agent expression is of the form

```
agent c.my_function(?, a, b)
```

where `a` and `b` are closed arguments (set at the time of the agent's definition), whereas `?` is an open argument, set at the time of any call to the agent. This agent is closed on the target `c`.

We can also define agents with an open target like

```
agent {C}.my_function(?, a, b)
```

where `{C}` denotes the class to which feature `my_function` belongs.

We implemented a class called `EIFFEL_PREDICATE[OBJECT_TYPE]` that inherits from `DB4O_PREDICATE` and `GENERICITY_HELPER`. It is to be initialized with an agent.

Class EIFFEL_PREDICATE[OBJECT_TYPE] has the following contract view:

```
class interface
   EIFFEL_PREDICATE[OBJECT_TYPE]

create
   make_open_target_agent,
   make_closed_target_agent

feature -- Match

   match(obj: OBJECT_TYPE): BOOLEAN
         -- Does `obj' match requirements?
         -- Uses either `open_target_predicate' or
         -- `closed_target_predicate' to
         -- decide for match result;
         -- Also tests whether `obj' conforms to
         -- `sample_object' if `sample_object' is generic.

invariant
   one_predicate: open_target_predicate /= Void xor
                  closed_target_predicate /= Void
   sample_not_void: sample_object /= Void

end -- class EIFFEL_PREDICATE
```

The return value of the match method is equal to the value of the agent (also note that match first tests whether candidate objects conform to sample_object if sample_object is generic, which means, we don't need to do the conformance test for generic objects when we are using EIFFEL_PREDICATE).

Suppose now that there is a Boolean function diagonal_greater_than(INTEGER) in the PARALLELOGRAM class, and we want to query for all parallelograms with diagonal greater than 10. Thanks to the agent mechanism, we don't need to define any new query method, but simply create an EIFFEL_PREDICATE instance and initialize it with

```
agent {PARALLELOGRAM}.diagonal_greater_than(10)
```

which is open on the target and closed on the argument. At run-time the target of the agent becomes the candidate object passed to the match method. To run the Native Query, we write

```
query_result := object_container.query(create
 {EIFFEL_PREDICATE[PARALLELOGRAM]}.make_open_target_agent(
       agent {PARALLELOGRAM}.diagonal_greater_than(10),
       create {PARALLELOGRAM}.make(1, 1)))
```

Agents also fit in the situations where the related class does not have a Boolean function corresponding to the query. In this case, we define a function in some class `MY_QUERY` like

```
diagonal_greater_than(PARALLELOGRAM; INTEGER): BOOLEAN
```

and initialize an `EIFFEL_PREDICATE` object with

```
agent a_query.diagonal_greater_than(?, 10)
```

which is closed on the target, open on the first and closed on the second argument. At run-time the first argument becomes the candidate object passed to the `match` method. To run the Native Query, we write

```
query_result := object_container.query(create
 {EIFFEL_PREDICATE[PARALLELOGRAM]}.make_closed_target_agent(
       agent a_query.diagonal_greater_than(?, 10),
       create {PARALLELOGRAM}.make(1,1)))
```

Note that if the developer uses `EIFFEL_PREDICATE` for the convenience of agents, he must endure some performance overhead caused by running agents. It was measured that `EIFFEL_PREDICATE` queries with open target agents run slower than `DB4O_PREDICATE` queries (that is queries that just inherit from `DB4O_PREDICATE` without using agents) by a factor of 1 – 3, and `EIFFEL_PREDICATE` queries with closed target agents run slower than `DB4O_PREDICATE` queries by a factor of 2 – 4. The more candidate objects, the more significant is the performance penalty.

3.3.5 Running Finalized Eiffel Assembly for Native Queries

Note that since the native query expression builder of db4o uses `Mono.Cecil.dll` to load the assembly which contains the `Predicate.Match` method in order to build an expression tree for the method, and `Mono.Cecil` seems to have some trouble with loading multi-module assemblies, the developer should always use the finalized Eiffel assembly to run Native Queries.

3.4 Query-By-Example vs. SODA Query API vs. Native Queries

Besides the limitations of Query-By-Example as described in db4o documentations, we have an additional limitation in Eiffel for .NET: Query-By-Example works only correctly if there are no descendant classes of the template class.

Native Queries are easy-to-use and type-safe, but at the moment they cannot be optimized to be run against indexes in Eiffel applications. So if performance is significant for the application, you may prefer SODA queries. Furthermore, avoiding feature renaming in descendant classes can prevent further performance loss when running a SODA query.

4 Configuration

Db4o provides a configuration interface to help developers fine-tune its behavior, such as setting the activation depth, the update depth, the delete behavior and the indexing for a class or for a field in a class. Because of the special mapping strategy of Eiffel types to .NET types, we must be careful when we are doing class-related and field-related configurations (recall that except for implementation classes of expanded types, all the other Eiffel implementation classes are direct subclasses of `System.Object`, and their fields are not inherited from any other classes). Three wrapper classes, `CONFIGURATION`, `OBJECT_CLASS` and `OBJECT_FIELD`, were implemented to make sure that once some configuration is done on a class `C` then the same configuration is also done on its descendant classes, and once some configuration is done on an attribute of a class `C`, then the same configuration is also done on the corresponding attributes of the descendant classes of `C`.

`CONFIGURATION` is the wrapper class for `ICONFIGURATION`, it is used for database-wide configurations, and its `object_class` function returns an `OBJECT_CLASS` object for configuration of the specified class.

```
object_class(clazz: SYSTEM_OBJECT): OBJECT_CLASS
        -- `OBJECT_CLASS' object to configure
        -- specified class
        -- `clazz' can be `TYPE[SYSTEM_OBJECT]',
        -- `SYSTEM_TYPE' or
        -- any other object used as a template.
    require
        clazz_not_void: clazz /= Void
```

`OBJECT_CLASS` is used for class-specific configurations, and its `object_field` function returns an `OBJECT_FIELD` object for configuration of the specified field.

```
object_field(fieldname: SYSTEM_STRING): OBJECT_FIELD
        -- `OBJECT_FIELD' object to configure
        -- specified field.
    require
        nonempty_fieldname:
          not {SYSTEM_STRING}.is_null_or_empty(fieldname)
```

`OBJECT_FIELD` is used for field-specific configurations.

4.1 Activation Depth

When objects are retrieved from the database, their fields are loaded into memory only to a certain depth, which is called “activation depth”. One must be careful when retrieving an object with a deep reference graph, because the default activation depth for any object is 5.

Suppose we have a class BTREE which represents a binary tree:

```
class
    BTREE

feature

    left, right: BTREE
    ...

end
```

Then we can use the db4o configuration interface to set the activation depth for BTREE. There are various ways to define the activation depth which applies during the whole database transaction: global, class-specific or field-specific; or we can also dynamically activate fields of retrieved objects.

4.1.1 Global Activation Depth

The following procedure shows you how to set the global activation depth for any object to 7:

```
configure_activation_depth is
    local
        config: CONFIGURATION
    do
        create config.make_global
        config.activation_depth_integer(7)
    end
```

4.1.2 Class-Specific Activation Depth

The following method calls show you how to set the minimum and maximum activation depth for the specified class and its descendant classes:

```
config.object_class({BTREE}).minimum_activation_depth_integer(7)
config.object_class({BTREE}).maximum_activation_depth(7)
```

With cascade activation of BTREE, the whole tree from the root to the leaves will be activated on retrieving. This setting can lead to increased memory consumption.

```
config.object_class({BTREE}).cascade_on_activate(True)
```

4.1.3 Field-Specific Activation

We can also automate activation for specific fields:

```
config.object_class({BTREE}).object_field(
                              "left").cascade_on_activate(True)
```

The above method call cascades activation of the `left` attribute of `BTREE` objects. However, as of this writing, the `IOBJECT_FIELD.cascade_on_activate` method does not work as specified, and the activation depth for `left` is still the default one after the method call.

4.1.4 Activating Fields Dynamically

The configurations described in section 4.1.1 – 4.1.3 are to be done before opening a database file and they apply during the next database transaction. On the other hand, we can also dynamically activate field references after retrieving objects from the database. The methods are defined in the `IOBJECT_CONTAINER` interface:

```
object_container.activate(object, depth)
```

To free some memory space, we can also deactivate fields using

```
object_container.deactivate(object, depth)
```

4.2 Update Depth

When we update an object, we must also pay attention to the update depth. The update depth for a class defines the number of levels of member objects which are to be updated automatically. The default update depth for all objects is 0, which means that `IOBJECT_CONTAINER.set(object)` method will only update the object passed as a parameter and any changes to its member objects will be lost. Db4o also provides different methods for setting the update depth.

4.2.1 Global Update Depth

The following procedure shows you how to set the global update depth for any object to 3:

```
configure_update_depth is
    local
        config: CONFIGURATION
    do
        create config.make_global
        config.update_depth(3)
    end
```

4.2.2 Class-Specific Update Depth

We can also specify the update depth for a certain class and its descendant classes:

```
config.object_class({BTREE}).update_depth(3)
```

or we can cascade update for a certain class and its descendant classes:

```
config.object_class({BTREE}).cascade_on_update(True)
```

4.2.3 Field-Specific Update Depth

The following method call cascades update for the specified field:

```
config.object_class({BTREE}).object_field(
                              "left").cascade_on_update(True)
```

4.2.4 Setting Update Depth Dynamically

The following method allows to dynamically set the update depth for a certain object:

```
object_container.ext.set(object, depth)
```

4.3 Delete Behavior

As db4o deletes only the object passed to `object_container.delete(obj)` with the referenced objects remaining in the database, we must define the delete behavior for a specific class or field to also delete the referenced objects.

4.3.1 Class-Specific Delete Behavior

The following method call cascades delete for class BTREE:

```
configure_delete_behavior is
    local
        config: CONFIGURATION
    do
        create config.make_global
        config.object_class({BTREE}).cascade_on_delete(True)
    end
```

4.3.2 Field-Specific Delete Behavior

The following method call cascades delete for attribute `left` of class BTREE:

```
config.object_class({BTREE}).object_field(
                              "left").cascade_on_delete(True)
```

We should keep in mind that there is no referential integrity check on delete.

4.4 Indexing

Indexing helps achieve maximum querying performance. Because of Eiffel's specific strategy of preserving multiple inheritance on the .NET platform, we have to pay attention when we are setting indexes for classes or fields.

4.4.1 Indexing of Classes

Let's continue with the example illustrated in section 2.3, indexing class PARALLELOGRAM means indexing PARALLELOGRAM, RHOMBUS, RECTANGLE and SQUARE. Using the wrapper class OBJECT_CLASS we can write:

```
configure_class_index is
    local
        config: CONFIGURATION
    do
        create config.make_global
        config.object_class({PARALLELOGRAM}).indexed(True)
    end
```

4.4.2 Indexing of Fields

Indexing attribute `height1` of class `PARALLELOGRAM` means indexing `PARALLELOGRAM.height1`, `RHOMBUS.height1`, `RECTANGLE.width` and `SQUARE.side_length`. Using the wrapper classes we only need to define field index once to get all the related fields indexed:

```
configure_field_index is
    local
        config: CONFIGURATION
    do
        create config.make_global
        config.object_class({PARALLELOGRAM}).object_field(
                                    “height1”).indexed(True)
    end
```

5 Refactoring

As application design changes in time, one of the challenges of object persistence is therefore keeping object databases up-to-date with the latest class schema without loss of old data. In the following sections we are going to examine db4o support for schema evolution.

5.1 Renaming Classes

Here is the procedure of how to configure the related db4o object container when a class is to be renamed:

1. Backup the database and application;
2. Close all open object containers if any;
3. Make a copy of the class to be renamed, and rename the class (do not remove old class yet);
4. Call `OBJECT_CLASS.rename_(new_name)` without having an object container open;
5. Open database file and close it again without actually working with it;
6. Remove the old class.

There is one issue related to renaming generic classes. Suppose we rename `GLIST[G]` as `GENERIC_LIST[G]`, then we should take two use cases into consideration:

- If the formal generic parameter `G` in the generic class `GLIST[G]` is only of reference type at run-time, then no additional work is to be done;
- If `G` becomes of expanded type, say `POINT`, at run-time, then we must also rename `GlistPoint` as `GenericListPoint`.

The conclusion for renaming generic classes is that we should call

```
OBJECT_CLASS.rename_(new_name)
```

once for all `GLIST[G]` instances with `G` being of a reference type, and once for each `G` of an expanded type.

5.2 Refactoring of Attributes

Attributes can be added or removed to or from a class, and they can also be assigned new types to. In the latter two cases, we need an additional API to be able to access values of the old attribute definition. Db4o provides this functionality through the interfaces `ISTORED_CLASS` and `ISTORED_FIELD`. We implemented two wrapper classes `STORED_CLASS` and `STORED_FIELD` to allow Eiffel developers to use class and attribute entities as defined in the code without the need of going into the .NET assembly to find out the corresponding .NET entity names.

There is one important point to be aware of: The functions in `STORED_CLASS` and `STORED_FIELD` only return information on the specified class and the specified attribute respectively. Descendant classes or attributes in descendant classes are not taken into consideration. The main justification is that as `STORED_CLASS` and `STORED_FIELD` represent metadata of classes and fields stored in the db4o database, they are static information. It may be better to have fine-grained information on class level.

On the other hand, the routines in the wrapper classes `OBJECT_CLASS` and `OBJECT_FIELD` configure the dynamic behavior of db4o databases, they are used to inform db4o databases about the whole class inheritance hierarchy in the way that configurations of descendant classes are also involved in there (see chapter 4 for details).

5.2.1 Adding Attributes

If you add a new attribute, db4o automatically starts storing the new data. Older instances of the stored class (from before the attribute was added) are still loaded, but the new attribute is set to its default value, or null. This is a dangerous choice, because the new class invariant may be invalidated by silently letting the object get into the system. A more correct choice would be to throw an exception, to force the developer to deal with the potential hassle using, for example, the appropriate callbacks to correctly initialize the newly added attributes.

5.2.2 Removing Attributes

If you remove an attribute, db4o ignores the stored value when activating instances of the class. The stored value is not removed from the database until the next defragment, and is still accessible via the `STORED_CLASS` / `STORED_FIELD` API.

5.2.3 Renaming Attributes

The following is the procedure of how to configure the related db4o object container when an attribute, say `att` in class `C`, is to be renamed as `new_att`:

1. Backup the database and application;
2. Close all open object containers if any;
3. Make a copy of class `C`, and name it differently, e.g. `C_OLD`;
4. Rename attribute `att` in `C` as `new_att`;
5. Call the following methods without having an object container open:
```
config.object_class({C}).object_field("att").rename_(
                                        {C_OLD}, "new_att")
```
6. Open the database file and close it again without actually working with it;
7. Remove class `C_OLD`.

This approach is slightly different from the one described in the db4o documentation. Here we need two class definitions to do attribute renaming: one is the class definition with the old attribute name, this helps us find out the corresponding .NET field name of the old attribute name; the other one is the class definition with the new attribute name, and with its help we can find out the corresponding .NET field name of the new attribute name. That's why we need to pass two arguments to the routine `rename_`, the first one being `SYSTEM_TYPE` of the old class, the second one being the new attribute name.

5.2.4 Changing Attributes' Types

If you modify an attribute's type, db4o internally creates a new attribute of the same name, but with the new type. The values of the old typed attribute are still present, but hidden. If you change the type back to the old type, the old values will still be there.

You can access the values of the previous attribute data using the `STORED_FIELD` API. In class `STORED_CLASS` the routine

```
stored_field(fieldname: SYSTEM_STRING;
             fieldtype: SYSTEM_OBJECT): STORED_FIELD
        -- Existing stored field of this stored class.
    require
        nonempty_fieldname:
            not {SYSTEM_STRING}.is_null_or_empty(fieldname)
        fieldtype_not_void: fieldtype /= Void
```

gives you access to the attribute whose type was changed.

In class STORED_FIELD, the routine

```
get(on_object: SYSTEM_OBJECT): SYSTEM_OBJECT
        -- Field value on `on_object'
    require
        on_object_not_void: on_object /= Void
```

returns the old attribute value for the specified object.

5.2.5 Changing Visibility of Attributes

In Eiffel for .NET, all the attributes and routines are compiled to public fields and public methods, even if the corresponding Eiffel features are exported to {NONE}. This leads to the fact that the change in visibility in an attribute does not have an impact on schema evolution.

5.3 Refactoring of Class Hierarchy

From the db4o documentation we learned that db4o does not directly support the following two refactorings:

- Inserting classes into an inheritance hierarchy;
- Removing classes from an inheritance hierarchy.

However, the above statement only holds for native .NET applications, it is not true for Eiffel for .NET applications.

As we know, in Eiffel for .NET the multiple inheritance of Eiffel types is realized through the multiple inheritance of .NET interfaces. With respect to refactoring this means:

- Inserting classes into an inheritance hierarchy is reduced to the refactoring issue of adding fields defined in the new classes to the classes implementing the subinterfaces;
- Removing classes from an inheritance hierarchy is reduced to the refactoring issue of removing fields defined in the removed classes from the classes implementing the subinterfaces.

6 Implementation

The classes implemented fall into five categories:

- Helper classes
    - `ATTRIBUTE_NAME_HELPER`
    - `GENERICITY_HELPER`
- Translator
    - `POINTER_TRANSLATOR`
- Wrapper classes for queries
    - `QUERY`
    - `CONSTRAINT`
    - `EIFFEL_PREDICATE`
- Wrapper classes for configurations
    - `CONFIGURATION`
    - `OBJECT_CLASS`
    - `OBJECT_FIELD`
    - `EIFFEL_CONFIGURATION`
- Wrapper classes for metadata of stored classes and fields
    - `STORED_CLASS`
    - `STORED_FIELD`

In the following sections we are going to give an introduction to each of these classes.

6.1 Helper Classes

Class `ATTRIBUTE_NAME_HELPER` provides routines for translation of Eiffel attribute names to the corresponding .NET field names, and class `GENERICITY_HELPER` provides routines for conformance test.

6.1.1 ATTRIBUTE_NAME_HELPER

Class ATTRIBUTE_NAME_HELPER is a framework class and it is not intended for direct use. It inherits from INTERNAL which is located in the assembly EiffelSoftware.Runtime.dll. INTERNAL gives access to the run-time type information of Eiffel objects and it also provides routines for mapping of Eiffel types to .NET types.

```
get_net_field_name(attrname: SYSTEM_STRING;
                   extent: SYSTEM_TYPE): SYSTEM_STRING
      -- The corresponding .NET field name for `attrname'
      -- in `extenttype'
   require
      attrname_not_void_or_empty:
         not {SYSTEM_STRING}.is_null_or_empty(attrname)
      extent_not_void: extent /= Void
   ensure
      result_not_void_or_empty:
         not {SYSTEM_STRING}.is_null_or_empty(Result)
```

get_net_field_name returns the corresponding .NET field name of an Eiffel attribute in a certain class. To do so, it traverses through the fields in the implementation class to find the field marked with the custom attribute EIFFEL_NAME_ATTRIBUTE whose value is equal to the name of the original Eiffel attribute; on the other hand, if there is already a field with the name equal to the first argument attrname, then the name of this field will be returned.

```
get_all_field_names(netfieldname: SYSTEM_STRING;
       extenttype: SYSTEM_TYPE): LINKED_LIST [SYSTEM_STRING]
      -- All related field names of `netfieldname',
      -- including `netfieldname' and its new names
      -- in implementation classes of `extenttype'
   require
      netfieldname_not_void_or_empty:
not {SYSTEM_STRING}.is_null_or_empty(netfieldname)
      extenttype_not_void: extenttype /= Void
   ensure
      result_not_void: Result /= Void
      no_empty_result: Result.count > 0
```

Given a field name in a certain implementation class and the interface the class implements, get_all_field_names returns a list of all the related field names which can be found in all the implementation classes of the interface. Continuing with the example introduced in section 2.3, the call

```
get_all_field_names("$$height1", {PARALLELOGRAM})
```

would return a list containing "$$height1", "$$width" and "$$sideLength".

```
get_descendant_field_name(netfieldname: SYSTEM_STRING;
 interface: SYSTEM_TYPE; destype: SYSTEM_TYPE): SYSTEM_STRING
       -- Field name corresponding to `netfieldname'
       -- in `destype'
    require
       eiffel_field: netfieldname.starts_with_string("$$")
       interface: interface.is_interface
       implementation: not destype.is_interface
    ensure
       result_not_void_or_empty:
             not {SYSTEM_STRING}.is_null_or_empty(Result)
```

get_descendant_field_name finds out the corresponding field name for `netfieldname' in the implementation class `destype' which implements `interface'. For a call

```
get_descendant_field_name("$$height1", {PARALLELOGRAM},
                              impl_rectangle)
```

where impl_rectangle is the System.Type object for the implementation class Impl.Rectangle, it would proceed as follows:

1. Find out the name of the getter method for the field $$height1, it is Height1;
2. Use INTERFACE_MAPPING to find out the name of the method in Impl.Rectangle which implements the Height1 method in interface Parallelogram, it is Width (see section 2.4.1 for details);
3. Convert Width to $$width, and $$width is the result of the function.

However, things become trickier when we have attributes of anchor types which are renamed in descendant classes. Suppose we have class GLIST[G] and class GSUBLIST[G] declared as follows:

```
class
   GLIST[G]

feature
   item: G
   next: like Current
...
end
```

```
class
   GSUBLIST[G]

inherit
   GLIST[G]
      rename
         item as subitem,
         next as subnext
      end
...
end
```

Then for a call

```
get_descendant_field_name("$$next", {GLIST[SYSTEM_OBJECT]},
                          impl_gsublistreference)
```

where `impl_gsublistreference` is the `System.Type` object for the implementation class `Impl.GsublistReference`, it would proceed as follows:

1. Find out the name of the getter method for the field `$$next`, it is `Next`;
2. Use `INTERFACE_MAPPING` to find out the name of the method in `Impl.GsublistReference` which implements the `Next` method in interface `GlistReference`, it is `_Next38` which is implemented as follows:

   ```
   .method public hidebysig newslot virtual
           instance class GlistReference
           _Next38() cil managed
   {
     .override GlistReference::Next
     // Code size      12 (0xc)
     .maxstack  8
     IL_0000:  ldarg.0
     IL_0001:  callvirt   instance class GsublistReference
                              Impl.GsublistReference::Subnext()
     IL_0006:  castclass  GlistReference
     IL_000b:  ret
   } // end of method GsublistReference::_Next38
   ```

   In the method body we see that `_Next38` simply calls the method `Subnext` in class `Impl.GsublistReference`. To extract the string `Subnext` from the method body, we use the Reflector for .NET assembly which allows to easily view, navigate, search, decompile and analyze .NET assemblies in C#, Visual Basic and IL.
3. Convert `Subnext` to `$$subnext`, and `$$subnext` is the result of the function.

```
is_eiffel_type(t: SYSTEM_TYPE): BOOLEAN
        -- Is `t' an Eiffel type?
    require
        t_not_void: t /= Void
```

An Eiffel class can only rename attributes inherited from an Eiffel class, and also there is no mapping done for an attribute name if the attribute is inherited from a non-Eiffel class. The routine `is_eiffel_type` can be used to check whether a type is an Eiffel type or not. It makes use of the fact that all the Eiffel types implement the interface `EIFFEL_TYPE_INFO`.

```
get_descendant_types(t: SYSTEM_TYPE): LINKED_LIST[SYSTEM_TYPE]
        -- A list of all implementation classes of `t'
    require
        t_not_void: t /= Void
    ensure
        result_not_void: Result /= Void
```

`get_descendant_types` returns a list of all the implementation classes of the given interface in the current application domain.

```
get_descendant_eiffel_types(t: SYSTEM_TYPE)
                              : LINKED_LIST[SYSTEM_TYPE]
        -- A list of all implementation classes of `t'
        -- which are defined in Eiffel assemblies
    require
        t_not_void: t /= Void
    ensure
        result_not_void: Result /= Void
```

`get_descendant_eiffel_types` returns a list of all the implementation classes of the given interface which are defined in the Eiffel assemblies loaded in the current application domain (Eiffel assemblies are marked with the attribute `EIFFEL_CONSUMABLE_ATTRIBUTE`).

6.1.2 GENERICITY_HELPER

Class `GENERICITY_HELPER` is used in SODA queries and Native Queries when generic objects or tuples are queries for, see section 3.2.3 and 3.3.2 for details. It inherits from class `INTERNAL` which provides detailed run-time type information of Eiffel objects. The public routines in `GENERICITY_HELPER` have the following contract view:

```
conforms_to_object(obj1: ANY; obj2: ANY): BOOLEAN
        -- Does `obj1' conform to `obj2'?
        -- The result takes conformance of generically derived
        -- types into account.
    require
        obj1_not_void: obj1 /= Void
        obj2_not_void: obj2 /= Void

get_conforming_objects(a_list: ILIST;
            an_object: SYSTEM_OBJECT): LIST[SYSTEM_OBJECT]
        -- List of objects in `a_list' which conform to
        -- `an_object'
    require
        list_not_void: a_list /= Void
        an_object_not_void: an_object /= Void
```

6.2 Translator

Db4o provides a way to specify a custom way of storing and retrieving objects through the `IOBJECT_TRANSLATOR` and `IOBJECT_CONSTRUCTOR` interfaces.

In Eiffel for .NET, there is a field named `$$____type` in every implementation class, it is used to store the run-time type information of actual generic parameters in case of a generic type. `$$____type` contains a `RuntimeTypeHandle` which has a field `Value` of type `System.IntPtr` which encapsulates a pointer to an internal data structure that represents the type. With the original db4o setting, `System.IntPtr` cannot be stored, it means `a_glist.$$____type.type.Value = 0` after retrieving and activating the `Impl.GlistReference` object. If we then access `a_glist.item`, an exception would be thrown with the following exception message:

```
Tag: Object reference not set to an instance of an object.
System.NullReferenceException: Object reference not set to an
instance of an object.
   at
EiffelSoftware.Runtime.Types.RT_CLASS_TYPE.conform_to(RT_TYPE
other)
```

The solution is a translator for `POINTER` which is the Eiffel counterpart for `System.IntPtr`. Since `RuntimeTypeHandle.Value` remains unchanged only within one application run, it makes no sense to store the pointer value. Class `POINTER_TRANSLATOR`, on storing an instance of `POINTER`, searches for the corresponding `TypeHandle` and then stores the type name and its containing assembly's location (i.e., the absolute path of the assembly) in the format of

```
full_name_of_type, location_of_assembly
```

On retrieving and activating of a certain `POINTER` object, the translator reads the type name and the assembly's location from the database, and initializes a valid `POINTER` object. With this approach, no exception is thrown when accessing a generic attribute of an object.

Class `POINTER_TRANSLATOR` inherits from interface `IOBJECT_CONSTRUCTOR` and implements the following four methods of the interface:

```
on_activate(container: IOBJECT_CONTAINER;
            application_object: SYSTEM_OBJECT;
            stored_object: SYSTEM_OBJECT)
      -- db4o calls this method during activation.

on_instantiate(container: IOBJECT_CONTAINER;
               stored_object: SYSTEM_OBJECT): SYSTEM_OBJECT
      -- Convert `stored_object' to a POINTER.
      -- db4o calls this method when `stored_object' needs
      -- to be instantiated.
   require else
      container_not_void: container /= Void
      stored_object_not_void: stored_object /= Void

on_store(container: IOBJECT_CONTAINER;
         application_object: SYSTEM_OBJECT): SYSTEM_OBJECT
      -- Convert the POINTER `application_object' to its
      -- corresponding type name and assembly's location
      -- in the format of
      -- `full_name_of_type, location_of_assembly'
      -- db4o calls this method during storage and query
      -- evaluation.
   require else
      container_not_void: container /= Void
      application_obj_not_void: application_object /= Void
   ensure then
      result_not_void: Result /= Void

stored_class: SYSTEM_TYPE
      -- {SYSTEM_STRING} converted to
```

Class POINTER_TRANSLATOR uses two hash tables for an efficient lookup: one has POINTER instances as key and type, assembly strings as value, and it is used on storing POINTER instances; the other one has type, assembly strings as key and POINTER instances as value, and it is used on instantiating POINTER objects. On initialization of a POINTER_TRANSLATOR instance, both hash tables are created and then filled with key-value pairs found in all the assemblies in the current application domain, and after that they are updated each time a new assembly is loaded into the current application domain.

Also note that since several database sessions may exist at the same time, and the translator including its hash tables is shared among the sessions (the database configuration is global), we need a thread-safe way to update the hash tables. Monitors are used for this purpose.

There is one disadvantage of storing type, assembly strings for a certain POINTER: once you move the assembly to another location, then POINTER_TRANSLATOR is not able to map the type, assembly string to the POINTER any more and an exception will be thrown (from within the routine POINTER_TRANSLATOR.on_instantiate), which means, retrieving generic objects would then fail. If you have to move an assembly to another location, you could implement your own translator similar to POINTER_TRANSLATOR to update the type, assembly strings.

6.3 Wrapper Classes for Queries

6.3.1 Wrapper Classes for SODA Query API

Two wrapper classes QUERY and CONSTRAINT are implemented to ensure correct query results of SODA queries for Eiffel objects (see section 3.2 for details).

Class QUERY exposes the following features:

```
query: IQUERY
        -- The actual db4o query object

executed: BOOLEAN
        -- Is query already executed?
```

```
constrain(constraint: SYSTEM_OBJECT): CONSTRAINT
        -- Add `constraint' to `Current' node and
        -- return a new `CONSTRAINT' for this query node or
        -- `Void' for objects implementing the `IEVALUATION'
        -- interface.
    require
        constraint_not_void: constraint /= Void
        not_executed: not executed

constraints: ICONSTRAINTS
        -- An `ICONSTRAINTS' object that holds an array of all
        -- constraints on this node.
    require
        not_executed: not executed

descend(eiffel_fieldname: SYSTEM_STRING;
          extenttype: SYSTEM_TYPE): QUERY
        -- A reference to a descendant node of
        -- `eiffel_fieldname' in the query graph
    require
        not_executed: not executed
        fieldname_not_void_or_empty:
not {SYSTEM_STRING}.is_null_or_empty(eiffel_fieldname)
    extenttype_not_void: extenttype /= Void

execute: IOBJECT_SET
        -- Execute query and return the result of query.
    require
        not_executed: not executed

order_ascending: QUERY
        -- Add an ascending order criteria to this node of the
        -- query graph and return `Current' to allow the
        -- chaining of method calls.
    require
        not_executed: not executed

order_descending: QUERY
        -- Add a descending order criteria to this node of the
        -- query graph and return `Current' to allow the
        -- chaining of method calls.
    require
        not_executed: not executed
```

```
sort_by(comparator: IQUERY_COMPARATOR): QUERY
        -- Sort the resulting `IOBJECT_SET' by `comparator'
        -- and return `Current' to allow the chaining of
        -- method calls.
    require
        not_executed: not executed
        comparator_not_void: comparator /= Void

optimize(config: ICONFIGURATION)
        -- Optimize `Current' and its child query nodes
        -- by setting index for fields involved.
```

Class CONSTRAINT exposes the following features:

```
constraint: ICONSTRAINT
        -- The actual ICONSTRAINT object for SODA query

and_(with: CONSTRAINT): CONSTRAINT
        -- Link `Current' with `with' for AND evaluation,
        -- return a new constraint, that can be used for
        -- further calls to `and_’ or `or_’.
    require
        with_not_void: with /= Void

by_example: CONSTRAINT
        -- Set the evaluation mode to object comparison (query
        -- by example), return `Current' to allow the chaining
        -- of method calls.

contains: CONSTRAINT
        -- Set the evaluation mode to containment comparison,
        -- return `Current' to allow the chaining of method
        -- calls.

ends_with(case_sensitive: BOOLEAN): CONSTRAINT
        -- Set the evaluation mode to string ends_with
        -- comparison, comparison will be case sensitive if
        -- `case_sensitive' is true,
        -- case insensitive otherwise,
        -- return `Current' to allow the chaining of method
        -- calls.
```

```
equal_: CONSTRAINT
        -- Used in conjunction with `CONSTRAINT.smaller' or
        -- `CONSTRAINT.greater' to create constraints like
        -- "smaller or equal", "greater or equal".
        -- Return `Current' to allow the chaining of method
        -- calls.

get_object: SYSTEM_OBJECT
        -- The `SYSTEM_OBJECT' the query graph was constrained
        -- with to create `Current'.

greater: CONSTRAINT
        -- Set the evaluation mode to ">",
        -- return `Current' to allow the chaining of method
        -- calls.

identity: CONSTRAINT
        -- Set the evaluation mode to identity comparison,
        -- return `Current' to allow the chaining of method
        -- calls.

like_: CONSTRAINT
        -- Set the evaluation mode to "like" comparison,
        -- return `Current' to allow the chaining of method
        -- calls.

not_: CONSTRAINT
        -- Turn on not_ comparison,
        -- return `Current' to allow the chaining of method
        -- calls.

or_(with: CONSTRAINT): CONSTRAINT
        -- Link `Current' with `with' for OR evaluation,
        -- return a new constraint, that can be used for
        -- further calls to `and_’ or `or_’.
    require
        with_not_void: with /= Void

smaller: CONSTRAINT
        -- Set the evaluation mode to "<",
        -- return `Current' to allow the chaining of method
        -- calls.
```

```
starts_with(case_sensitive: BOOLEAN): CONSTRAINT
        -- Set the evaluation mode to string starts_with
        -- comparison, comparison will be case sensitive if
        -- `case_sensitive' is true,
        -- case insensitive otherwise,
        -- return `Current' to allow the chaining of method
        -- calls.
```

The usage of SODA queries for Eiffel objects is almost the same as the usage of SODA queries for native .NET objects, except for the routine `descend`. If there is only one `descend("a_field_name")` in a query, then we can rely on the type information of the extent specified for the query to find out the .NET name of `a_field_name`. Problems arise when we have more than one `descend` in sequence, in these cases field name mapping cannot be done if run-time type information for the class containing the field lacks in the query wrapper. Here are some use cases illustrating the issue:

- **Using non-Eiffel .NET classes**

  Suppose there is a .NET class `Stack` with a field `top` of type `System.Object`, then an Eiffel for .NET application uses it to stack `CAR` objects and then stores several `Stack`s in the database. In a query, the client wants to get all `Stack`s whose top car is a Ferrari.

  ```
  constraint := query.constrain({STACK})
  subquery := query.descend("top").descend("model")
  ```

  Since the .NET reflection mechanism returns `System.Object` as the type of `Stack.top`, how do we know `top` is a `CAR` object at runtime? And how to get the .NET name of `model` in `CAR`?

- **Genericity**

  Suppose there is a generic class `GLIST[G]` with `item: G`, and its client uses it as `GLIST[CAR]`. In the .NET assembly, `GLIST[G]` becomes `GlistReference` and `item` becomes of type `System.Object`. The question is still how to map `model` of `CAR` to its .NET name?

  ```
  constraint := query.constrain({GLIST[CAR]})
  subsubquery := query.descend("item").descend("model")
  ```

  Since tuples are all generic (every tuple has an attribute `$$nativeArray` which is of type `object[]`), this question is also relevant when querying for tuples.

- **Type redeclaration in descendant classes**

  Suppose we have PARALLELOGRAM with `center: POINT_2D` and its descendant class PARALLELOGRAM_3D which redeclares `center` as `center: POINT_3D` (POINT_3D inherits from POINT_2D). we want to query for PARALLELOGRAM objects whose `z` is smaller than 5:

  ```
  constraint := query.constrain({PARALLELOGRAM})
  subsubquery := query.descend("center").descend("z")
  ```

  How to find the .NET name for `z`?

To make the above queries possible, we implemented the `descend` routine with the following signature where the first argument is the attribute name, and the second argument is the SYSTEM_TYPE object of the class which contains the specified attribute.

```
descend(eiffel_fieldname: SYSTEM_STRING;
        extenttype: SYSTEM_TYPE): QUERY
      -- A reference to a descendant node of
      -- `eiffel_fieldname' in the query graph
   require
      not_executed: not executed
      fieldname_not_void_or_empty:
not {SYSTEM_STRING}.is_null_or_empty(eiffel_fieldname)
      extenttype_not_void: extenttype /= Void
```

Using the wrapper classes QUERY and CONSTRAINT, the queries would be

- **Using non-Eiffel .NET classes**

  ```
  constraint := query.constrain({STACK})
  subquery := query.descend("top", {STACK})
  subsubquery := subquery.descend("model", {CAR})
  subsubconstraint := subsubquery.constrain("Ferrari")
  queryresult := query.execute
  ```

- **Genericity**

  ```
  constraint := query.constrain({GLIST[CAR]})
  subquery := query.descend("item", {GLIST[CAR]})
  subsubquery := subquery.descend("model", {CAR})
  subsubconstraint := subsubquery.constrain("Ferrari")
  queryresult := query.execute
  ```

- **Type redeclaration in descendant classes**

```
constraint := query.constrain({PARALLELOGRAM})
subquery := query.descend("center", {PARALLELOGRAM})
subsubquery := subquery.descend("z", {POINT_3D})
subsubconstraint := subsubquery.constrain(5).smaller
queryresult := query.execute
```

This approach seems a little redundant, but it manages to solve the problem.

6.3.2 Wrapper Class for Native Queries

Class `EIFFEL_PREDICATE[OBJECT_TYPE]` is the wrapper class for `DB4O_PREDICATE`, meant to allow Eiffel developers using agents for Native Queries. See section 3.3.4 for a detailed explanation of the class.

6.4 Wrapper Classes for Configurations

6.4.1 Global Configuration for Eiffel Applications

In class `EIFFEL_CONFIGURATION`, database settings for Eiffel applications are configured globally for all the db4o transactions. It has the following contract view:

```
class interface
    EIFFEL_CONFIGURATION

create
    configure

feature -- Configuration
    configuration: CONFIGURATION
        -- Global configuration for db4o transactions

    configure
        -- Do global configuration for db4o transactions.

    install_translators
        -- Install translator for POINTER.

end -- class EIFFEL_CONFIGURATION
```

Before opening any db4o transaction, make sure that you have called the following method in your client application

```
configure_global is
    local
        c: EIFFEL_CONFIGURATION
    do
        create c.configure
    end
```

In the current version of EIFFEL_CONFIGURATION, only POINTER_TRANSLATOR is installed. We also wanted to install TypeHandlers for Eiffel strings, i.e. classes STRING and STRING_32, however, as of this writing, TypeHandlers do not work correctly yet, so Eiffel developers are encouraged to implement TypeHandlers for Eiffel strings and install them in EIFFEL_CONFIGURATION once TypeHandlers work correctly.

Custom TypeHandlers let you control the way objects are stored to the database and retrieved in a query. See http://developer.db4o.com/Resources/view.aspx/Reference/Implementation_Strategies/TypeHandlers for details.

6.4.2 CONFIGURATION

Class CONFIGURATION is the wrapper class for ICONFIGURATION, and it exposes the same methods as ICONFIGURATION, except for object_class which has the following signature:

```
object_class(clazz: SYSTEM_OBJECT): OBJECT_CLASS
        -- `OBJECT_CLASS' object to configure the specified
        -- class. `clazz' can be `TYPE[SYSTEM_OBJECT]',
        -- `SYSTEM_TYPE' or an object used as a template.
    require
        clazz_not_void: clazz /= Void
```

IOBJECT_CLASS is used to configure dynamic behavior of how db4o deals with objects stored in the database, such as activation depth, update depth, delete behavior, indexing, etc. Configurations of the descendant classes must therefore also be involved. That's why we implemented a wrapper class for IOBJECT_CLASS, which is called OBJECT_CLASS, so that the Eiffel developers only need to call configurations on an interface, say {PARALLELOGRAM}, to have all the implementation classes of Parallelogram configured accordingly, that is, Impl.Parallelogram, Impl.Rhombus, Impl.Rectangle and Impl.Square (see the example in section 2.3).

6.4.3 `OBJECT_CLASS` and `OBJECT_FIELD`

Class `OBJECT_CLASS` exposes the same methods as `IOBJECT_CLASS`, except for `object_field` which has the following signature:

```
object_field(fieldname: SYSTEM_STRING): OBJECT_FIELD
        -- `OBJECT_FIELD' object to configure the specified
        -- field.
    require
        nonempty_fieldname: not
{SYSTEM_STRING}.is_null_or_empty(fieldname)
```

Class `OBJECT_FIELD` exposes the same methods as `IOBJECT_FIELD` except for `rename_` which has the following signature (see section 5.2.3 for an explanation of the routine):

```
rename_(old_eiffel_type: SYSTEM_TYPE;
          new_name: SYSTEM_STRING)
        -- Rename this field as `new_name'.
    require
        nonempty_new_name: not
{SYSTEM_STRING}.is_null_or_empty(new_name)
```

For configurations of implementation classes of a certain interface, we create and initialize an `OBJECT_CLASS` instance with a `SYSTEM_TYPE` object of the interface, and then call the corresponding configuration method to configure the implementation classes accordingly. For example,

```
local
    config: CONFIGURATION
do
    create config.make_global
    config.object_class({PARALLELOGRAM}).update_depth(2)
end
```

sets the update depth of `Impl.Parallelogram`, `Impl.Rhombus`, `Impl.Rectangle` and `Impl.Square` to 2.

For configurations of some field in the implementation classes of a certain interface, we create and initialize an `OBJECT_FIELD` instance with an `OBJECT_CLASS` instance and a field name, and then call the corresponding configuration method to configure the fields (including those renamed in the descendant classes) in all the implementation classes of the interface. For example,

```
  local
     config: CONFIGURATION
     oc: OBJECT_CLASS
  do
     create config.make_global
     oc := config.object_class({PARALLELOGRAM})
     oc.object_field("height1").indexed(True)
  end
```

sets field indexes for the fields Impl.Parallelogram.$$height1, Impl.Rhombus.$$height1, Impl.Rectangle.$$width and Impl.Square.$$sideLength.

6.5 Wrapper Classes for Metadata of Stored Classes and Fields

6.5.1 STORED_CLASS

Class STORED_CLASS is the wrapper class for ISTORED_CLASS. It provides metadata information of the stored classes.

STORED_CLASS exposes the same methods as ISTORED_CLASS except for stored_field which has the following signature:

```
  stored_field(fieldname: SYSTEM_STRING;
               fieldtype: SYSTEM_OBJECT): STORED_FIELD
        -- Existing stored field of this stored class.
     require
        nonempty_fieldname: not
  {SYSTEM_STRING}.is_null_or_empty(fieldname)
        fieldtype_not_void: fieldtype /= Void
```

A STORED_CLASS instance is initialized with an interface type or a template object, and it only provides metadata information of the direct implementation class of the interface, in case of {PARALLELOGRAM}, we only get information about the class Impl.Parallelogram, no information is given about Impl.Rhombus or Impl.Rectangle, etc.

6.5.2 STORED_FIELD

Class STORED_FIELD is the wrapper class for ISTORED_FIELD. It provides metadata information of the stored fields in a stored class.

STORED_FIELD exposes the same methods as ISTORED_FIELD except for rename_ which has the following signature:

```
rename_(old_eiffel_type: SYSTEM_TYPE; new_name: SYSTEM_STRING)
        -- Rename this field as `new_name'.
    require
        nonempty_new_name: not
{SYSTEM_STRING}.is_null_or_empty(new_name)
```

Class STORED_FIELD only gives information about the field in the direct implementation class of a specified interface. Fields in other implementation classes of the interface are not involved. For example,

```
local
    sc: STORED_CLASS
    sf: STORED_FIELD
do
    create sc.make(object_container.ext, {PARALLELOGRAM})
    sf := sc.stored_field("height1", {INTEGER})
    sf.create_index
end
```

only creates an index for the field Impl.Parallelogram.$$height1, but not for the fields Impl.Rhombus.$$height1, Impl.Rectangle.$$width or Impl.Square.$$sideLength.

7 Using .NET Delegates

Db4o enables the client to add listeners to an `IOBJECT_CONTAINER` for the following events

- QueryStarted
- QueryFinished
- Creating (first time an object is about to be saved)
- Created (after the object is saved)
- Activating
- Activated
- Deactivating
- Deactivated
- Updating
- Updated
- Deleting
- Deleted
- Committing
- Committed

These callbacks can be used to gather statistics information, to perform validity or constraints check and stop the execution if necessary, or to initiate some special behavior after the action has been taken.

For `QueryStarted` and `QueryFinished` events the client registers `QueryEventHandler` delegate:

```
public delegate void QueryEventHandler(
           Object sender, QueryEventArgs args)
```

For `Creating`, `Activating`, `Deactivating`, `Updating` and `Deleting` events the client registers `CancellableObjectEventHandler` delegate:

```
public delegate void CancellableObjectEventHandler(
           Object sender, CancellableObjectEventArgs args)
```

For `Created`, `Activated`, `Deactivated`, `Updated` and `Deleted` events the client registers `ObjectEventHandler` delegate:

```
public delegate void ObjectEventHandler(
           Object sender, ObjectEventArgs args)
```

For `Committing` and `Committed` events the client registers `CommitEventHandler` delegate:

```
public delegate void CommitEventHandler(
           Object sender, CommitEventArgs args)
```

Delegates are supported in Eiffel for .NET. For example, in a C# program we write the following code to register an `OnCreated` event handler to container:

```
IObjectContainer OpenObjectContainer()
{
   try {
      IObjectContainer db = Db4oFactory.OpenFile("f.db4o");
      IEventRegistry registry =
EventRegistryFactory.ForObjectContainer(db);
      registry.Created += new ObjectEventHandler(OnCreated);
      return db;
   } catch (Exception ex) {
   }
   return null;
}

static void OnCreated(object sender, ObjectEventArgs args)
{
   // handling code
}
```

The Eiffel for .NET counterpart is

```
open_object_container: IOBJECT_CONTAINER is
   local
      registry: IEVENT_REGISTRY
      handler: OBJECT_EVENT_HANDLER
   do
      Result := {DB_4O_FACTORY}.open_file("f.db4o")
      registry :=
{EVENT_REGISTRY_FACTORY}.for_object_container(Result)
      create handler.make(Current, $on_created)
      registry.add_created(handler)
   rescue
      Result := Void
   end
```

```
on_created(sender: SYSTEM_OBJECT; args: OBJECT_EVENT_ARGS) is
    do
        -- handling code
    end
```

The Eiffel's specific mechanism for objects which represent operations, called agents, is more powerful than .NET delegates because of its support for open and closed arguments and for open and closed target. However, agents are not compatible with .NET delegates (all agents are of type `FUNCTION` or `PROCEDURE`, while all delegates are descendants of the `DELEGATE` class), which means for the above example,

```
create handler.make(Current, $on_created)
registry.add_created(handler)
```

cannot be replaced with

```
// compile-time error: non-conforming actual argument in
// feature call
registry.add_created(agent on_created(?, ?))
```

Conclusion: we can only use delegates, not agents, to register db4o callbacks.

8 Cross Compatibility between C# and Eiffel for .NET

Since Eiffel for .NET and C# are both languages supported on the .NET Framework, it might be interesting to see whether objects stored with an Eiffel application can be retrieved by a C# application and vice versa. The answer is yes, except for code involving genericity and tuples.

8.1 Using Eiffel to Retrieve C# Objects

An Eiffel application needs two things to be able to retrieve C# objects: the assembly which contains the class definitions of the objects and the database file.

There is nothing special with regard to querying for classes, structs and interfaces. We just write queries as we would do in a C# program. For example, to retrieve all `PARALLELOGRAM` instances whose `_height1` is greater than 10, we write:

```
retrieve is
    local
        db: IOBJECT_CONTAINER
        closed: BOOLEAN
        query: IQUERY
        c: ICONSTRAINT
        resultos: IOBJECT_SET
    do
        db := {DB_4O_FACTORY}.open_file("NetObjects.db4o")
        query := db.query
        c := query.constrain (({PARALLELOGRAM}).to_cil)
        c := query.descend("_height1").constrain (10).greater
        resultos := query.execute
        closed := db.close
    rescue
        if (not closed and then db /= Void) then
            closed := db.close
        end
    end
```

`({PARALLELOGRAM}).to_cil` is the Eiffel notation to get the type of `PARALLELOGRAM`. Queries for structs or interfaces are done in a similar way.

Eiffel for .NET does not consume .NET generics, so if there are .NET generic objects stored in a database file, then you cannot retrieve them in an Eiffel for .NET application.

8.2 Using C# to Retrieve Eiffel Objects

To get correct query results of Eiffel objects, it is important to

1. Install `PointTranslator` before opening the database, see section 6.2 and 6.4 for details;

2. Always query for interfaces both in SODA and Native Queries because the multiple inheritance hierarchy structure of Eiffel types is preserved through the multiple inheritance hierarchy structure of .NET interfaces;

3. Use the helper classes, especially `QUERY` and `CONSTRAINT` for SODA queries (see section 3.2 for reasons described in detail).

The following is an example which shows how to retrieve all instances of `Parallelogram` whose `height1` is greater than 10 (`Parallelogram` is an interface):

```
void Retrieve()
{
    IObjectContainer db =
          Db4oFactory.OpenFile("eiffel_objects.db4o");
    try
    {
        Query eiffelQuery =
          Db4oForEiffel.Create.Query.MakeFromQuery(db.Query());
        Constraint eiffelConstraint =
          eiffelQuery.Constrain(typeof(Parallelogram));
        Query eiffelSubquery =
          eiffelQuery.Descend("height1", typeof(Parallelogram));
        Constraint eiffelSubconstraint =
          eiffelSubquery.Constrain(10).Greater();
        IObjectSet resultos = eiffelQuery.Execute();
    }
    finally
    {
        db.Close();
    }
}
```

It is difficult to get a correct query result for an Eiffel generic type or tuple in C#. In Eiffel we have the helper class `GENERICITY_HELPER` which takes advantage of the Eiffel for .NET run-time system to get the type information of actual generic parameters of an Eiffel generic object. To tell whether a candidate object is of the right (generic) type, the routines in

GENERICITY_HELPER need a sample object which provides the necessary type information of actual generic parameters. For example, the object

```
  create {GLIST[PARALLELOGRAM]}
```

has PARALLELOGRAM as its first actual generic parameter. However, on the C# side, we don't have a straightforward way to create an Eiffel generic object with the necessary type information of the actual generic parameters. That means, GENERICITY_HELPER cannot be used for conformance tests in C# applications, and we may get non-conforming objects in a query result for Eiffel generic objects.

9 Persistence of C Structs

Eiffel for .NET can be used to persist C structs in db4o databases.

First, we define a struct in, say, `point.h`:

```
typedef struct {
   int x;
   int y;
} Point;
```

Second, we implement an Eiffel wrapper class `POINT` for `struct Point`:

```
class
   POINT

create
   make,
   make_with_x_y

feature {NONE} -- Initialization
   make is
         -- Creation method
      do
         create internal_item.make(structure_size)
      end

   make_with_x_y(a_x, a_y: INTEGER) is
         -- Initialize Current with `a_x' and `a_y'.
      do
         make
         set_x(a_x)
         set_y(a_y)
      end

feature -- Command
   set_x(a_x: INTEGER) is
         -- Set `x' with `a_x'.
      do
         c_set_x(item, a_x)
         x := a_x
      ensure
         set: x = a_x
      end
```

```
    set_y(a_y: INTEGER) is
            -- Set `y' with `a_y'.
        do
            c_set_y(item, a_y)
            y := a_y
        ensure
            set: y = a_y
        end

feature -- Query
    structure_size: INTEGER is
            -- Size of Current structure.
        do
            Result := c_size_of_point
        end

    x: INTEGER
            -- x position

    y: INTEGER
            -- y position

    item: POINTER is
            -- Pointer to C struct
        do
            Result := internal_item.item
        ensure
            not_void: Result /= default_pointer
        end

feature {NONE} -- Implementation
    internal_item: MANAGED_POINTER
            -- Managed pointer to the struct.

feature {NONE} -- C externals
    c_size_of_point: INTEGER is
            -- Point struct size.
        external
            "C [macro %"point.h%"]"
        alias
            "sizeof(Point)"
        end
    c_set_x(a_item: POINTER; a_x: INTEGER) is
            -- Set `a_item''s x with `a_x'
        external
            "C inline use %"point.h%""
        alias
            "[
            {
                ((Point *)$a_item)->x = (EIF_INTEGER)$a_x;
            }
            ]"
        end

    c_set_y(a_item: POINTER; a_y: INTEGER) is
            -- Set `a_item''s y with `a_y'
        external
            "C inline use %"point.h%""
        alias
            "[
            {
                ((Point *)$a_item)->y = (EIF_INTEGER)$a_y;
            }
            ]"
        end

    c_x(a_item: POINTER): INTEGER is
             -- `a_item''s x
        external
            "C inline use %"point.h%""
        alias
            "[
                ((Point *)$a_item)->x
            ]"
        end

    c_y(a_item: POINTER): INTEGER is
            -- `a_item''s y
        external
            "C inline use %"point.h%""
        alias
            "[
                ((Point *)$a_item)->y
            ]"
        end

end
```

Note that in POINT we have two attributes x: INTEGER and y: INTEGER which correspond to int x and int y in struct Point. We need this duplication to make sure that besides the memory location of a Point, its x- and y-coordinates are also stored in the database.

Third, store POINT instances as usual:

```
store is
         -- Store `POINT' instances.
    local
        p: POINT
    do
        create p.make_with_x_y(1, 2)
        object_container.store(p)
    end
```

To retrieve all the POINT instances whose x is greater than 1, we write

```
retrieve is
         -- Retrieve `POINT' instances.
    local
        query: IQUERY
        constr: ICONSTRAINT
        resultos: IOBJECT_SET
        p: POINT
    do
        query := object_container.query
        constr := query.constrain(({POINT}).to_cil)
        constr := query.descend("$$x").constrain (1).greater
        resultos := query.execute
    end
```

({POINT}).to_cil is the Eiffel notation for the run-time type of POINT.

10 Conclusions and Future Work

Eiffel is well integrated in the .NET Framework. The multiple inheritance hierarchy structure of Eiffel classes is preserved, and feature adaptation techniques also work well in the .NET run-time system. However, there are some issues around Eiffel genericity.

Eiffel applications can use all db4o features, but we must be careful when querying for Eiffel objects and when doing class- or field-related configurations.

While in case of querying for non-generic objects, SODA and Native queries return objects according to the Eiffel's conformance rule, in case of querying for generic objects, developers must take over the task of filtering out non-conforming objects using the helper class we implemented.

A wrapper for SODA Query API, taking care of all the aforementioned issues, has also been developed.

Using agents for Native Queries makes db4o very appealing to Eiffel developers, though the performance overhead may sometimes be significant. The db4o team is making a big effort to optimize Native Queries so that they can be run against indexes. Eiffel applications, however, cannot take advantage of the optimization algorithms yet. The open question is therefore whether and how Native Queries in Eiffel applications can be optimized to be run against indexes.

Wrapper classes are implemented for class- and field-related configurations. They ensure a consistent database behavior when activating, updating, deleting and indexing objects in the sense that configurations are done on all the descendant classes of a specified class.

Wrapper classes are also implemented for metadata information of stored classes and fields. Using them the Eiffel developer can use the original Eiffel class names or feature names to get the related information.

Db4o supports refactoring while some approaches may be inappropriate.

As for the cross compatibility between Eiffel and other .NET (e.g., C#) applications, we can say that we can use Eiffel to retrieve C# objects and vice versa except for generic objects.

Having a native implementation of an object-oriented database for the Eiffel programming language would be nice so that some or all the issues encountered could be avoided.

References

1. Meyer, B.: Object-Oriented Software Construction, 2nd edition. Prentice Hall, 1997.
2. Smacchia, P.: Practical .NET2 and C#2. Paradoxal Press, 2005.
3. .NET Developer Center, http://msdn2.microsoft.com/en-us/library/aa139615.aspx
4. Simon, R., Stapf, E., Meyer, B.: Full Eiffel on the .NET Framework, http://msdn2.microsoft.com/en-us/library/ms973898.aspx
5. Db4o documentation, http://developer.db4o.com/Resources/view.aspx/Documentation
6. Project web site: http://developer.db4o.com/ProjectSpaces/view.aspx//Defcon
7. Reflector for .NET: http://www.aisto.com/roeder/dotnet/

Appendix Getting Started with Db4o for Eiffel

In this appendix we show you how to use db4o databases within Eiffel applications.

Step 1: Download Db4o Assembly

Because of a critical bug in the db4o assemblies before version 7.1.26 (see section 3.2.5), it is important that you download a db4o version later than 7.1.26.

Db4o download center:

http://developer.db4o.com/files/default.aspx

Step 2: Download Reflector for .NET Assembly

In the implementation we take advantage of the Reflector for .NET assembly to examine Eiffel for .NET assemblies. You can download the assembly at

http://www.aisto.com/roeder/dotnet/

or

https://svn.origo.ethz.ch/defcon/source_code/references/Reflector.exe

Step 3: Download Source Code of Db4o for Eiffel

The project is implemented in Eiffel, and its source code is to be imported into your project. The source code is available at

https://svn.origo.ethz.ch/defcon/source_code/db4o_for_eiffel.zip

After downloading the zip file, please extract the file and then move the `db4o_for_eiffel` directory to the directory in which your project will reside, say `db4o_example`.

Step 4: Download EiffelStudio

It is common that Eiffel developers use EiffelStudio for developing Eiffel applications. You can download it at

https://www2.eiffel.com/download/

or at

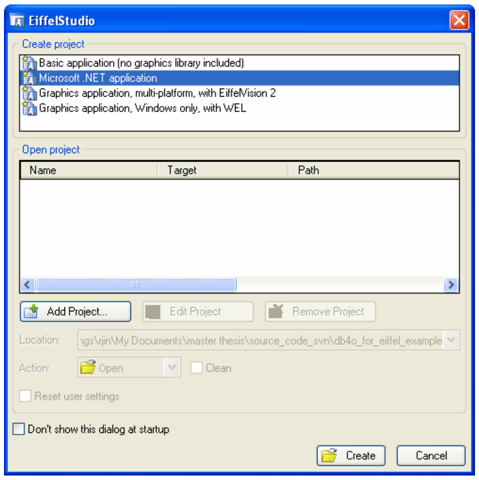
http://eiffelstudio.origo.ethz.ch/download

The recommended version for Windows XP is EiffelStudio 6.1.6.9962, other versions of 6.1.x seem to have problems with debugging Eiffel for .NET applications.

Step 5: Create an EiffelStudio Project

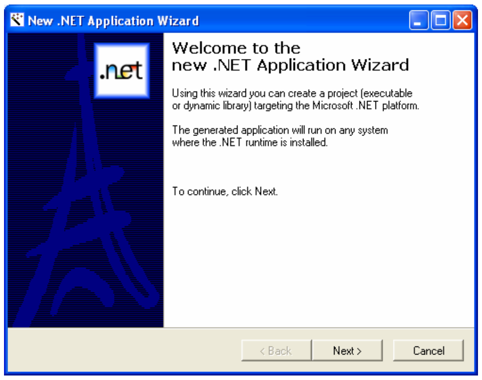
Until now you should have a project directory called `db4o_example` with the `db4o_for_eiffel` directory in it.

Start EiffelStudio, and in the pop-up window select "Microsoft .NET application" and then click "Create" (see screenshot).


EiffelStudio
Create project
Basic application (no graphics library included)
Microsoft .NET application
Graphics application, multi-platform, with EiffelVision 2
Graphics application, Windows only, with WEL
Open project
Name
Target
Path
Add Project...
Edit Project
Remove Project
Action:
Open
Clean
Reset user settings
Don't show this dialog at startup
Create
Cancel

Location: ıgs\rjin\My Documents\master thesis\source_code_svn\db4o_for_eiffel_example

Click "Next >" to continue.



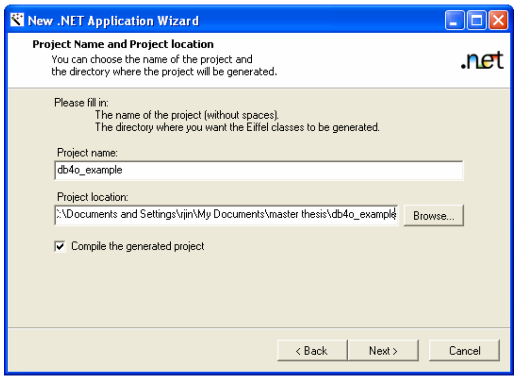
Welcome to the new .NET Application Wizard

Using this wizard you can create a project (executable or dynamic library) targeting the Microsoft .NET platform.

The generated application will run on any system where the .NET runtime is installed.

To continue, click Next.

In the next window specify the project name as “db4o_example” and the project location as the location of your `db4o_example` directory. Click “Next >”.



Project Name and Project location

You can choose the name of the project and the directory where the project will be generated.

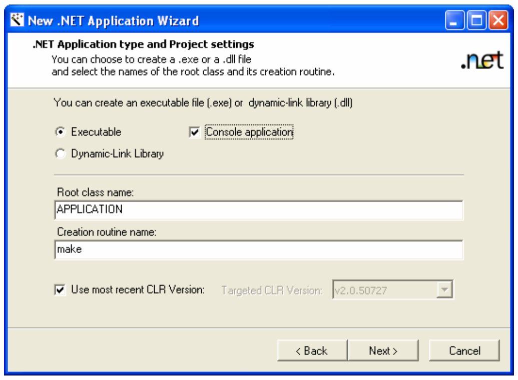
Please fill in:
The name of the project (without spaces).
The directory where you want the Eiffel classes to be generated.

Project name: db4o_example

Project location: :\Documents and Settings\rjin\My Documents\master thesis\db4o_example

[✓] Compile the generated project

Configure the .NET application as shown in the following screenshot:



.NET Application type and Project settings

You can choose to create a .exe or a .dll file and select the names of the root class and its creation routine.

You can create an executable file (.exe) or dynamic-link library (.dll)

(●) Executable [✓] Console application

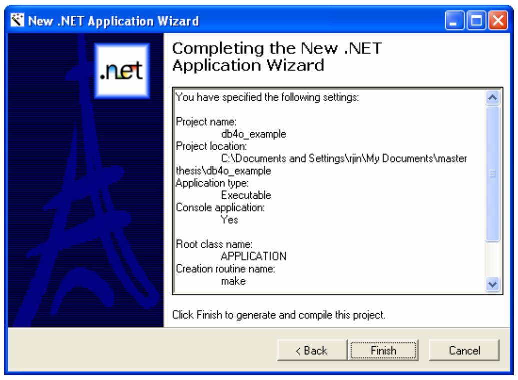
( ) Dynamic-Link Library

Root class name: APPLICATION

Creation routine name: make

[✓] Use most recent CLR Version: Targeted CLR Version: v2.0.50727

Click “Finish” to generate and compile the project.



Completing the New .NET Application Wizard

```
You have specified the following settings:

Project name:
        db4o_example
Project location:
        C:\Documents and Settings\rjin\My Documents\master
thesis\db4o_example
Application type:
        Executable
Console application:
        Yes

Root class name:
        APPLICATION
Creation routine name:
        make
```

Click Finish to generate and compile this project.

Step 6: Add Assemblies to Your Project

In the “Clusters” panel right-click “Assemblies” and in the context-menu select “Add Assembly ...”. Then in the “Add Assembly” window specify the location of the db4o assembly `Db4objects.Db4o.dll` to import it into the project.

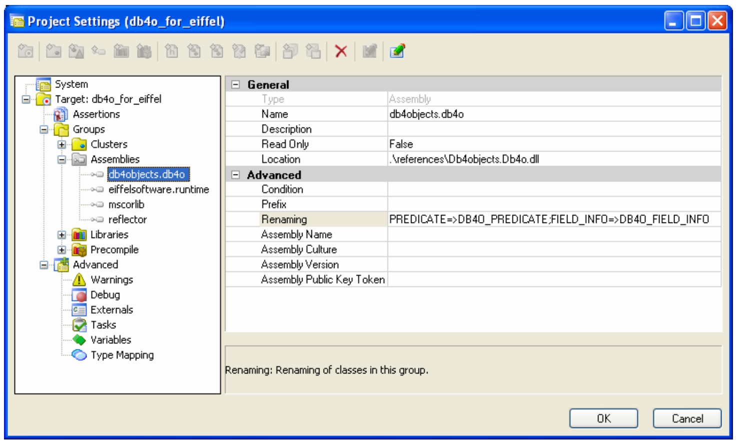
Perform the same steps for the Reflector for .NET assembly `Reflector.exe`.

We also need the `EiffelSoftware.Runtime` assembly which can be selected in the “Add Assembly” window.

Step 7: Rename Classes to Avoid Name Clashes

In Eiffel for .NET every class must have a unique class name, so we have to rename some classes in `Db4objects.Db4o.dll` and `Reflector.exe` to avoid name clashes.

Click “Project” in the menu bar and select “Project settings ...”.

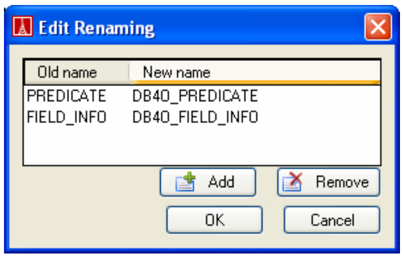


Project Settings (db4o_for_eiffel)

General	
Type	Assembly
Name	db4objects.db4o
Description	
Read Only	False
Location	.\references\Db4objects.Db4o.dll
Advanced	
Condition	
Prefix	
Renaming	PREDICATE=>DB4O_PREDICATE;FIELD_INFO=>DB4O_FIELD_INFO
Assembly Name	
Assembly Culture	
Assembly Version	
Assembly Public Key Token	

Renaming: Renaming of classes in this group.

In the “Project Settings” window, navigate to the “db4objects.db4o” item in the left panel, then in the right panel click inside the cell next to “Renaming”. A window for editing renaming then pops up. Rename `PREDICATE` as `DB4O_PREDICATE` and rename `FIELD_INFO` as `DB4O_FIELD_INFO`.



Edit Renaming

Old name	New name
PREDICATE	DB4O_PREDICATE
FIELD_INFO	DB4O_FIELD_INFO

Similar for the `Reflector.exe` assembly, rename `ASSEMBLY` as `REFLECTOR_ASSEMBLY` and rename `ICONFIGURATION` as `REFLECTOR_ICONFIGURATION`.

Step 8: Configure Db4o Databases for Eiffel Applications

Before storing and querying for Eiffel objects in db4o databases, we have to install `POINTER_TRANSLATOR`, which is done in the class `EIFFEL_CONFIGURATION` (see section 6.2 for details). Add the following method in the `APPLICATION` class and call it in the root procedure `make`.

```
init is
        -- Set global database configuration.
    local
        eiffel_configuration: EIFFEL_CONFIGURATION
    do
        create eiffel_configuration.configure
    end

make is
        -- Run application.
    do
        init
        ...
    end
```

Step 9: Open and Close a Db4o Database

To open and close a db4o database, add the following features to class `APPLICATION`:

```
feature  -- Database control

    db: IOBJECT_CONTAINER

    database_file: STRING is "eiffel.db4o"

    open_database is
            -- Open `db' of `database_file'.
        do
            db := {DB_4O_FACTORY}.open_file(database_file)
        end

    close_database is
            -- Close `db'.
        local
            closed: BOOLEAN
        do
            closed := db.close
        end
```

Step 10: Store Eiffel Objects

Suppose we have a class `PARALLELOGRAM` with two attributes `height1` and `height2`:

```
class
   PARALLELOGRAM

create
   make

feature {NONE} -- Initialization

   make(h1: INTEGER; h2: INTEGER) is
         -- Initialize `height1' with `h1',
         -- `height2' with `h2'.
      require
         h1_positive: h1 > 0
         h2_positive: h2 > 0
      do
         height1 := h1
         height2 := h2
      end

feature -- Access

   height1: INTEGER
   height2: INTEGER

end
```

To store some `PARALLELOGRAM` objects we can write

```
store is
   local
      closed: BOOLEAN
   do
      open_database
      db.store(create {PARALLELOGRAM}.make(10, 30))
      db.store(create {PARALLELOGRAM}.make(20, 40))
      close_database
   rescue
      if (db /= Void) then
         closed := db.close
      end
   end
```

Step 11: Retrieve Eiffel Objects

Db4o supplies three querying mechanisms: Query-By-Example, SODA Query API and Native Queries.

Query-By-Example

The following query uses Query-By-Example to retrieve all the `PARALLELOGRAM` objects whose `height1` is equal to 10 and `height2` is equal to 30:

```
retrieve_qbe is
    local
        template: PARALLELOGRAM
        resultos: IOBJECT_SET
        closed: BOOLEAN
    do
        open_database
        create template.make(10, 30)
        resultos := db.query_by_example(template)
        printos(resultos)
        close_database
    rescue
        if (db /= Void) then
            closed := db.close
        end
    end
```

where `printos` outputs the `PARALLELOGRAM` objects in the query result to the console:

```
printos(os: IOBJECT_SET) is
    local
        p: PARALLELOGRAM
    do
        from
        until not os.has_next
        loop
            p ?= os.next
            if (p /= Void) then
                io.put_string("Parallelogram (" + p.height1.out
+ ", " + p.height2.out + ")")
                io.put_new_line
            end
        end
    end
```

SODA Query API

To retrieve all `PARALLELOGRAM` objects whose `height1` is greater than 10, you can write the following SODA query:

```
retrieve_soda is
   local
      query: QUERY
      constraint, subconstraint: CONSTRAINT
      resultos: IOBJECT_SET
      closed: BOOLEAN
   do
      open_database
      create query.make_from_query(db.query)
      constraint := query.constrain({PARALLELOGRAM})
      subconstraint := query.descend("height1",
{PARALLELOGRAM}).constrain(10).greater
      resultos := query.execute
      printos(resultos)
      close_database
   rescue
      if (db /= Void) then
         closed := db.close
      end
   end
```

Native Queries

You should first define a class, say `PARALLELOGRAM_PREDICATE`, which inherits from `DB4O_PREDICATE` and implements the `match` method. The `match` method defines whether a candidate object is to be included in the query result or not.

```
class
   PARALLELOGRAM_PREDICATE

inherit
   DB4O_PREDICATE

feature
   match(p: PARALLELOGRAM): BOOLEAN is
      do
         Result := p.height1 > 10
      end

end
```

Then you can pass a `PARALLELOGRAM_PREDICATE` instance to the `IOBJECT_CONTAINER.query` method to get the query result:

```
retrieve_nq is
    local
        resultos: IOBJECT_SET
        closed: BOOLEAN
    do
        open_database
        resultos := db.query(create {PARALLELOGRAM_PREDICATE})
        printos(resultos)
        close_database
    rescue
        if (db /= Void) then
            closed := db.close
        end
    end
```

Note that you have to run the finalized system to have Native Queries run without exceptions.

Step 12: What's Next

To ensure a correct way of working with db4o databases within Eiffel applications, you may need to consult the previous chapters of this document. Furthermore, we suggest you to download and try out the following advanced example project when you are reading this documentation:

https://svn.origo.ethz.ch/defcon/source_code/db4o_for_eiffel_example.zip

For further advanced features of db4o, such as transaction and concurrency control, maintenance, client-server mode, etc. please visit the db4o documentation page:

http://developer.db4o.com/Resources/