Eiffel for .NET Binding for Db4o

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

Abstract. 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.

The effort that this paper describes is the implementation of the necessary Db4o framework classes to make it usable within Eiffel applications.

Keywords: Db4o, Query-By-Example, SODA Queries, Native Queries, Eiffel for .NET, multiple inheritance, genericity, agents

1 Introduction

Eiffel is a pure object-oriented programming language that since a long time implements 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 the project is to identify peculiarities of persisting Eiffel objects and to provide solutions so that Eiffel developers can use db4o as seamlessly as possible.

In section 2 we give an overview of how Eiffel types are mapped to .NET types without loosing the multiple inheritance hierarchy information. In section 3 we discuss the three db4o querying mechanisms, showing how we can adapt them for querying Eiffel objects. In section 4 we present our considerations on mapping Eiffel generic types to .NET types, and in section 5 we draw some conclusions.

2 Mapping from Eiffel Types to .NET Types

Since the common language runtime of the .NET Framework only supports single inheritance, 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, if we compile the Eiffel classes `RHOMBUS`, `RECTANGLE` and `SQUARE`, with `SQUARE` inheriting from both `RHOMBUS` and `RECTANGLE`, we have three .NET types generated for each effective (fully implemented) Eiffel class, say `SQUARE`: the interface `Square`, the class `Impl.Square` which implements the interface `Square`, and the class `Create.Square` whose static methods are used to create and initialize instances of `Impl.Square`. The `Square` interface extends the two interfaces `Rhombus` and `Rectangle`. As a matter of fact, in .NET all instances of Eiffel types are related through interfaces and they don't directly inherit from each other. 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.

3 Querying for Eiffel Objects using db4o

Db4o supplies three querying systems: Query-By-Example, the 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

Since in .NET all instances of Eiffel types are related through interfaces and they don't directly inherit from each other, 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

```
template_object := create {RECTANGLE}.make(10, 0)
query_result := object_container.get(template_object)
```

only returns direct instances of `RECTANGLE` of width=10, while `SQUARE` objects are not returned.

Query-By-Example has several limitations, and only fits with very simple queries. Although a wrapper class for Query-By-Example could be implemented which returns not only the direct instances but also instances of subclasses, we are not going to implement it because of its restricted functionalities.

3.2 SODA Query API

The SODA Query API provides Eiffel applications with an efficient, though type-unsafe way to query for objects. When building a query graph, a SODA Query descends to a field by specifying the field name, which the Eiffel compiler then converts. The following query specifies the interface `{RECTANGLE}` as the query extent to find all `RECTANGLE` (including `SQUARE`) objects whose width equals 10:

```
query := object_container.query
constraint := query.constrain({RECTANGLE})
subconstraint := query.descend("$$width").constrain(10)
query_result := query.execute
```

Apart from the ugly prepended "$$", another issue with SODA queries originates from renaming a feature in subclasses. In our example, if we rename the `width` feature to `side_length` in the `SQUARE` class, then the above query will not return `SQUARE` objects, because the `Impl.Square` class only has a field called `$$sideLength` instead of `$$width`. To retrieve a correct query result, we have to modify the query as follows:

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

To make SODA queries in Eiffel as easy as in other .NET applications, we are implementing a wrapper for the SODA query API which does the field name translation and gets the field constraints or-ed if the field is renamed in subclasses.

3.3 Native Queries

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

In the .NET version of db4o the developer can use delegates as 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 introduced a class called `EIFFEL_PREDICATE` that inherits from `PREDICATE`, and has to be initialized with an agent. The return value of its `match` method is equal to the value of the agent. Suppose now that there is a Boolean function `diagonal_greater_than(INTEGER)` in the `RECTANGLE` class, and we want to query for all rectangles with diagonal greater than 20. 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 {RECTANGLE}.diagonal_greater_than(20)
```

using an agent which is open on the target and closed on the argument. At run-time the target of the agent will be the candidate object passed to the `match` method.

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(RECTANGLE; INTEGER): BOOLEAN
```

and initialize the `EIFFEL_PREDICATE` object with

```
agent a_query.diagonal_greater_than(?, 20)
```

, an agent which is closed on the target, open on the first and closed on the second argument. The downside of Native Queries in Eiffel applications is that they cannot be optimized. Since the `match` method in subclasses of `PREDICATE` is declared like

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

and `candidate` becomes of an interface type `SomeType` at run-time, db4o cannot optimize interface method calls in the `match` method body, because has to instantiate objects of `SomeType`, run the code and then decide the query result. Furthermore, 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 `PREDICATE` queries (that is queries that just inherit from `PREDICATE` without using agents) by a factor of 1 – 3, and `EIFFEL_PREDICATE` queries with closed target agents run slower than `PREDICATE` queries by a factor of 2 – 4.

4 Genericity

An Eiffel generic type is mapped to the .NET types as follows:

```
class                          class GlistReference
   GLIST[G]                    {
feature                           object item;
   item: G                     }
end
```

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, and this is the reason why `GLIST[STRING]` objects would also be returned for a query for `GLIST[RECTANGLE]` objects.

To solve this issue, we implemented a helper class which understands whether two (generic) objects conform to each other according to Eiffel's conformance rule, which states that 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.

5 Conclusions and future work

Eiffel applications can use all db4o features. 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 Queries, taking care of all the aforementioned issues, is also under development. 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.

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