Master Thesis

Multi-format, EiffelStudio-integrated object browser and writer

Author[s]:
Hansen, Lucien

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005816802

Rights / License:
In Copyright - Non-Commercial Use Permitted
Multi-Format, EiffelStudio-integrated Object Browser and Writer

Master Thesis

By: Lucien Hansen
Supervised by: Marco Piccioni
Prof. Dr. Bertrand Meyer

Student Number: 03-917-044
Abstract

Many formats are nowadays used to store objects for later use. In general, formats can be divided into binary (privileging efficiency) and textual (for easier comprehension and manipulation), XML being an example of the latter. At the moment the Eiffel language is using exclusively a binary format.

The first goal of this thesis is to build a new Eiffel persistence library for reading and writing objects in dADL, a textual, human-readable format. The library is also integrated into "PERSIST", a project about devising a unified persistence framework for Eiffel. Finally, it offers, as an additional feature, custom serialization, namely the possibility to choose which objects to store.

The second goal is to extend the capability of Ebbro, an already existing object browsing tool, by integrating the capability to read objects coded in dADL and write them back to storage.

The third goal is to integrate the new Ebbro into EVE, the EiffelStudio research branch. This will allow Eiffel developers to use it directly from the programming environment.
Acknowledgments

I would like to thank Marco Piccioni for supporting me during my thesis. I also very much appreciated his constant valuable feedback. I also want to thank Thomas Beale and Peter Gummer for their feedback during this thesis. My thanks of course also to Prof. Bertrand Meyer for giving me the opportunity to write this master thesis at the Chair of Software Engineering.

Finally I want to thank my family and close friends for always supporting me during my time at the Swiss Federal Institute of Technology ETH Zurich.
1 Introduction

Object persistence is an important factor in modern programming languages. The possibility
to store information from objects and then retrieve the information again, is a critical and
important operation. A lot of programs depend on object persistence.

To serialize this object information we need a specific format. In the Eiffel programming
language [11] there currently only exists the binary format, which is not human readable, but
a very compact and efficient format in the eyes of a program. So the first focus of this thesis
is to provide a new Eiffel library to serialize and deserialize an object. The format of this new
library is dADL [1], which is similar to XML [10]. The advantages of this new format are for
one that it is human readable and as a result also allows human beings to look at a file inside
any common text-editor. On the other hand since the dADL format is not Eiffel specific, it
opens doors to more easily transform this format into other formats (for example to XML) and
also makes the objects more portable between applications.

A big part of this library has already been implemented at openEHR [2]. So this part of the
thesis is built upon the following project: http://www.openehr.org/projects/eiffel.html.

Ebbro [6] is a standalone graphical tool. The user can open an object file from file-system
and then is able to browse through the serialized object (in the binary format). It also allows a
user to update simple values inside an object and then this new information can be serialized
back to the file.

The second goal of this thesis is to use the new dADL library inside Ebbro. This should result
in having the same functionality also for objects stored in the dADL format.

But since Ebbro is a standalone tool at the moment the work of this thesis also includes
integrating Ebbro into EVE [5], an EiffelStudio [9] research branch. This will allow a
developer to use Ebbro from within the Eiffel development environment.

I will shortly add a few words about how this document is organized.

Chapter 2 lists the requirements for this master thesis. In Chapter 3 the first part concerning
the dADL library is explained in detail. In Chapter 4 the second part of the thesis regarding
the Ebbro tool is documented. In Chapter 5 and 6 challenges and issues concerning both the
dADL library and Ebbro are listed. In Chapter 7 I will conclude my thesis and look at future
work.

At the end of this document there is also a short glossary.
2 Requirements

2.1 Intended Results
In the first part of the thesis the existing dADL library implementation is extended to handle structures and shared objects. The project targets version 6.3 of the Eiffel compiler integrated into EiffelStudio. In addition, a "custom" serialized form should be possible, that is, being able to choose programmatically which attributes to save. Afterwards to guarantee reusability, serializer/deserializer classes should also be integrated in the persistence framework [4].

In part II the result should be a fully integrated application which uses the library described in part I and provide an easy to use interface for browsing serialized objects including the ability to change some elementary fields and store them back. The objects are either in the dADL format or in binary format.

For both parts of the project the development language is Eiffel [11].

2.2 Objectives and Priorities
This part will list all the objectives and priorities for this master thesis:

- Update the Eiffel language version used for the dADL serializer to 6.3 (high priority)
- Integrate the classes into the persistence framework (medium priority)
- Complete the class \textit{DT\_OBJECT\_CONVERTER} [7] for structures (high priority)
- Complete the class \textit{DT\_OBJECT\_CONVERTER} for shared objects (high priority)
- Implement a “custom” serialized form to choose programmatically which attributes to save (medium priority)
- Integrate the new dADL library into EVE (high priority)
- Interface to browse objects in dADL format with EVE (high priority)
- Interface to browse objects in binary format with EVE (high priority)
- Interface which allows changing elementary fields (numeric expanded fields, strings, booleans) for objects in dADL format (medium priority)
- Interface which allows changing elementary fields (numeric expanded fields, strings, booleans) for objects in binary format (medium priority)
- Perform extensive and documented tests for correctness (medium priority)
- Perform performance tests (medium priority)
3 dADL Library

3.1 Overview

The dADL library is a persistence library for the programming language Eiffel. The library is based on an openEHR project [3].

As a text-based serial format dADL (ADL (Archetype Definition Language) [1]) is used. An example of this format is given in section 3.6.

The library is here to serialize/deserialize Eiffel objects in the dADL format.

This project is being hosted on Origo [8] and can be accessed via this project page: http://dadle.origo.ethz.ch/wiki/dadle.

For more documentation or people who are interested in the source code, I would recommend visiting the project page.

Here is a simple example of the dADL format:

```
(EXAMPLE_OBJECT) <
  my_string = (STRING_8) "shared"
  my_shared_str = (STRING_8) </my_str>
  list_string = (LINKED_LIST<STRING_8>) "string1", "string2"
  my_integer = (INTEGER_32) <200>
>
```

3.2 Background

The project started at openEHR and this thesis now uses the code-base produced by Thomas Beale and Peter Gummer. The code used inside the library is based on the SVN directory [12] from November 2008 (Revision number: 779).

This code-base includes a parser and the main classes to build up the data type structure. But it misses functionality to serialize/deserialize structures (e.g. list types, array types, tuples...) and it doesn’t handle shared objects.

3.3 Implementation

This part gives an overview of the implementation.

The dADL specification can be found in this document:

(Chapter 4. dADL - Data ADL)

The complete source code of this dADL library can be accessed on the Origo SVN directory:

http://svn.origo.ethz.ch/wsvn/dadle/.
3.3.1 Dependencies
The dADL library depends on other Eiffel libraries. The dependencies are shown in the following diagram:

Figure 1  dADL library dependencies

3.3.2 Persistence Framework
The library was integrated into the persistence framework [4]. The PERSIST project is about developing a single, integrated and simple persistence framework for Eiffel.

The library is thus well structured and has a predefined interface. The persistence framework classes for the dADL format are the following:

- **DADL_PERSISTENCE_MANAGER**
  It inherits from the base class in the framework PERSISTENCE_MANAGER and provides the main mechanisms used to serialize/deserialize objects.

- **DADL_FORMAT**
  It inherits from the base class in the framework PERSISTENCE_FORMAT and is responsible for serializing/deserializing in the dADL format.

The complete bon diagram is shown in Figure 2:
3.3.3 Parser
The library uses a gobo parser generated with the following files:
dadl_scanner.l and dadl2_validator.y.
\[
\text{[openEHR\%}
\text{ref_impl_eiffel\%libraries\%common_libs\%src\%structures\%syntax\%dadl\%parser\%]}
\]
The parser is then being generated with the help of gobo binaries: gelex and geyacc.

3.3.4 Serializing / Deserializing
The main class here is DT\_OBJECT\_CONVERTER.
This class is responsible for converting an Eiffel object into a dADL dt_structure and vice versa. The class is also able to convert a dt_structure which corresponds to an Eiffel object type, which is not known to the system. In this case the dt\_to\_object method will simply return an object of type DADL\_DECODED. As seen in Figure 2 the class DADL\_DECODED inherits from GENERAL\_DECODED, which class specifies common functionality for a decoded object (such as an attribute list with all the attributes and the class name...).

The mechanism to also serialize/deserialize objects, whose types are not known to the system, was developed from scratch in this thesis, since the original project did only work with known types.

The implementation for serializing a DADL\_DECODED object is pretty straightforward. The attribute list gets traversed and every attribute then serialized according to the already implemented or added features for serializing known types. If an attribute is also of unknown type a recursion starts. This recursion will terminate, because every object is based on a known type to the system sooner or later in the hierarchy.
For deserializing an unknown object the implementation is a bit trickier. The main routine here is ‘decode_dt_from_unknown_object’. With the help of the class name the implementation figures out, what kind of object it is dealing with (e.g. generic, \textit{TUPLE}, \textit{SPECIAL}…). The object will then get decoded accordingly by analyzing the dt_structure. For every unknown object a \textit{DADL_DECODED} object will be created, which holds the attributes and other information about the object. If an attribute is of a known type to the system, it can be deserialized using the existing mechanisms for known types, otherwise a recursion starts. But this recursion will also terminate, because every unknown object will at some point only have known attribute types. Explaining the whole mechanism in detail would go to far here. For interested readers I recommend looking at the source code, which is also documented.

\subsection*{3.3.5 Shared Objects}

The library is able to handle shared objects as described in the description of the dADL archetype language \cite{1} (4.4.6 Associations and Shared Objects).

The \texttt{DT_OBJECT_CONVERTER} object remembers all objects which get serialized/deserialized - ‘remember_object(an_obj:ANY;a_path:STRING)’. In this way before serializing an object, it is checked whether the current object has already been serialized or not. So in case the object has already been serialized, now only the path to this object gets serialized (as described in the dADL description).

The same thing is used, just in the opposite direction, for deserializing. So once the method \texttt{dt_to_object} encounters an \texttt{OG_PATH} object, it will deserialize the object at this path in case the object has not yet been deserialized - otherwise the object can simply be taken out of the ‘obj_storage’.

This is an example of a dADL file with shared objects:

\begin{verbatim}
(SHARED_EXAMPLE) <
  shared = (MY_SHARED) <
    some_string = (STRING_8) <"test"> some_int = (INTEGER_32_REF) <5>
  >
  my_str = (STRING_8) <"shared”>
  my_shared_str = (STRING_8) </my_str>
  my_seq = (LINKED_LIST<MY_SHARED>) <
    ["1"] = (MY_SHARED) </shared>
  >
>
\end{verbatim}
3.3.6 Void Attributes

Void attributes are also being serialized. There was also an issue about whether one should serialize or not serialize void attributes. For more information about this please read the issue in 6.1.

The syntax used is \(<void>\). The DT_PRIMITIVE_OBJECT class is able to carry void values and the DT_OBJECT_CONVERTER serializes/deserializes void attributes.

This is an example of a dADL file with void attributes:

```
(TC_VOID) <
    my_string_list = (LINKED_LIST<STRING>) "test_str",void,void,"string"
    my_hash = (HASH_TABLE<STRING,STRING>) <
        ["key1"] = (STRING) "test"
        ["key2"] = (NONE) <void>
    >
    my_tuple = (TUPLE<A_CLIENT,STRING,STRING,INTEGER_32>) <
        ["1"] = (NONE) <void>
        ["2"] = (NONE) <void>
        ["3"] = (STRING) "test"
        ["4"] = (INTEGER_32) <100>
    >
>
```

3.3.7 Custom Serialization

The library also allows you to create a custom serialization form. This basically means you are able to choose which attributes from an object you want to serialize. This is a useful feature for two reasons:

- By decreasing the number of attributes you are serializing you can significantly improve the performance and of course decrease the overhead of serializing information in general.
- By devising a logical serialized form (instead of mirroring the physical representation) you can free the serialized form from being tightly coupled to the internal object representation. As a consequence, the serialized form can be much more robust to modifications of the internal representation itself over time. (Joshua Bloch, Effective Java, Addison Wesley 2001)

As for the implementation: In the DT_OBJECT_CONVERTER class the method 'object_to_dt' is called and as a parameter a_serialized_form_list:ARRAYED_LIST[STRING] can be provided. This list would specify which attributes to serialize. If the list is void, simply all attributes will be stored.

This implementation ensures that the object which gets serialized and the custom serialized form for this class are not coupled. So the implementation is persistence-agnostic.
3.4 Usage

3.4.1 Serialization
Serialization of an Eiffel object in the dADL format is achieved by using the class 
*DADL_PERSISTENCE_MANAGER* and then calling 'store (an_object: ANY)'.

*Note:* Also take a look at the preferences you are able to set in the dadl persistence manager 
class (like serializing type information or not).

3.4.2 Custom Serialization
You can create a custom serialized form, which specifies what attributes to serialize from a 
given class, in the following way:

- Create a *SERIALIZED_FORM* object.
- Add your custom serialized form to this object ("put_serialized_form") [By providing a 
  list of attribute names you want to store].
- Before serializing an object for which you created a serialized form, add your 
  *SERIALIZED_FORM* object to the *DADL_FORMAT* [By using 
  "persistence_manager.format.set_serialized_form..."].

From now on if you serialize an object with your persistence manager object - your custom 
serialized form will be applied to the objects it applies to.
Sample code

```eiffel
custom_serialize_example is
  -- example code to serialize an object with a custom serialized form
  local
    manager: DADL_PERSISTENCE_MANAGER
    custom_form: SERIALIZED_FORM
    object: MY_CLASS1
    attribute_list: ARRAYED_LIST[STRING]
  do
    -- create object to store
    create object

    -- specify which attributes to serialize
    create attribute_list.make (3)
    attribute_list.extend ("attribute_name1")
    attribute_list.extend ("attribute_name2")

    -- create custom serialized form object and add our
    -- attribute_list for 'MY_CLASS1'
    create custom_form.make
    custom_form.put_serialized_form (attribute_list, object.generator)

    -- create persistence_manager and add our serialized_form
    create manager.make ("c:/my_class1.adls")
    manager.format.set_serialized_form (custom_form)

    -- serialize 'object' to the filesystem
    manager.store (object)
  end
```

3.4.3 Deserialization

Deserialization of an Eiffel object in the dADL format is achieved by using the class `DADL_PERSISTENCE_MANAGER` and then calling 'retrieve: ANY'.

**Note:** The library is also able to deserialize an object file, which contains an Eiffel object, whose type is unknown to the system, which deserializes it. So the dADL library can actually not reconstruct the object and return it. But the library will put all the information given in the object file into an object of type `DADL_DECODED` and return this object as a result of "retrieve". As a developer one could then access the attribute values through the `DADL_DECODED` interface.

3.5 Performance Tests

The tests compare the dADL library to other serialization mechanisms in Eiffel.

The serialization and deserialization of Eiffel objects is being tested. So far only the times between binary, dADL and C serialization mechanism are measured. C serialization is very similar to the binary serialization mechanism. It was basically used to have one more serialization mechanism to compare the results with.
3.5.1 Test Description

A few facts about the tests:

- Hardware: Intel (R) Core(TM)2 Duo CPU T7500 @ 2.20GHz 3.00 GB RAM
- Test Program: A finalized Eiffel binary has been used (assertions were discarded, exception trace is disabled and inlining of size 0 is used).
- Binary serialization: The independent store mechanism is used (gc is disabled and without optimization for retrieval).
- dADL serialization: This library is used (The main class `DT_OBJECT_CONVERTER` is accessed through the dADL persistence framework classes (`DADL_FORMAT`...)).
- C serialization: on `RAW_FILE` - ‘independent_store’ is used.
- Optimizations: No code optimizations so far.
- Timing: The timing includes file access - so the objects get serialized to a file and are deserialized from the file again. I use a RAM disk, to minimize the file access overhead.

3.5.2 Evaluation

The Test results and diagrams can be viewed at the end of this document in the appendix.

Serializing

The dADL serializing mechanism is a two digit factor slower than the binary/C serializing mechanism. The C serializing mechanism seems to be the fastest of the three.

The difference gets bigger the more attributes are being serialized and it also depends on the complexity of the serialized attributes. So the more complex the serialized attributes are, the bigger is the time difference.

Deserializing

The picture is pretty much the same as for serializing, but the time difference between dADL and binary/C deserializing seems to be a bit smaller than for serializing. In deserializing the binary mechanism seems to be the fastest of the three mechanisms.

The results and graphs can be viewed at the end of this document in the appendix.
3.6 Example
This is an example of an Eiffel object serialized in the dADL format:

```
(EXAMPLE_OBJECT) <
    list_string = (LINKED_LIST<STRING_8>) <"string1", "string2", "test">
    list_custom = (LINKED_LIST<CUSTOM>) <
        ["1"] = (CUSTOM) <
            my_string = (STRING_8) <"test">
            my_array = (ARRAY<Integer_32>) <
                ["1"] = (Integer_32) <1>
                ["2"] = (Integer_32) <2>
                ["3"] = (Integer_32) <3>
            >
            my_int = (Integer_32_REF) <100>
        >
    >
    ht_primitive = (HASH_TABLE<Integer_32,STRING_8>) <
        ["key1"] = (Integer_32) <1>
        ["key2"] = (Integer_32) <2>
    >
    tuple_prim = (TUPLE<Integer_32,STRING_8,BOOLEAN>) <
        ["1"] = (Integer_32) <1>
        ["2"] = (STRING_8) <"test">
        ["3"] = (BOOLEAN) <True>
    >
    tuple_custom = (TUPLE<CUSTOM,LINKED_LIST<STRING_8>>) <
        ["1"] = (CUSTOM) <
            my_string = (STRING_8) <"test">
            my_array = (ARRAY<Integer_32>) <
                ["1"] = (Integer_32) <1>
                ["2"] = (Integer_32) <2>
                ["3"] = (Integer_32) <3>
            >
            my_int = (Integer_32_REF) <100>
        >
        ["2"] = (LINKED_LIST<STRING_8>) </list_string>
    >
    string = (STRING_8) <"test %&quot;&/()=?`^!{}[\]^°">
    char = (CHARACTER_8_REF) <"\"">
    bool = (BOOLEAN_REF) <True>
    nat_16 = (NATURAL_16_REF) <21>
    natural = (NATURAL_32_REF) <30>
    real = (REAL_32_REF) <0.234>
>
```

Short explanation:
- "list_string" : attribute name
- (LINKED_LIST<STRING_8>) : attribute type
- <"string1", ...> : attribute value
3.7 dADL vs. Binary format

This part will shortly illustrate the fundamental differences between the dADL and binary format. But one cannot say that one format is better than the other, they are just completely different and both have advantages. But nevertheless it should illustrate that there is a need for more than one format to serialize objects.

Here is the same object in dADL and in binary format:

- **dADL:**

```plaintext
(MY_CLASS2) <
  my_string = (STRING_8) "test_string"
  my_string2 = (STRING_8) "test_string2"
  parent_string = (NONE) void
  my_client = (A_CLIENT) <
    my_string = (STRING_8) "client_string1"
    my_string2 = (STRING_8) "client_string2"
    my_int = (INTEGER_32) 50
    my_bool = (BOOLEAN) True
  >
  my_int = (INTEGER_32) 12
  my_bool = (BOOLEAN) False
>
```

- **binary (not complete):**

```
MY_CLASS2:MY_CLIENT [SPECIAL [CHARACTER_8] STRING_8]
  STRING_8 my_string = "test_string"
  STRING_8 my_string2 = "test_string2"
  NONE parent_string
  A_CLIENT my_client:
    STRING_8 my_string = "client_string1"
    STRING_8 my_string2 = "client_string2"
    INTEGER_32 my_int = 50
    BOOLEAN my_bool = True

(MY_CLIENT) <
  my_string = (STRING_8) "client_string1"
  my_string2 = (STRING_8) "client_string2"
  my_int = (INTEGER_32) 50
  my_bool = (BOOLEAN) True
>
```

(This is how a serialized binary object looks like in a normal text-editor.)
4 Ebbro

4.1 Overview
Ebbro is a graphical tool to help developers or users to view and browse through serialized objects. The tool displays the information, which is stored inside an object in a tree view. Each attribute’s name, value and type information is being displayed.

Ebbro has more functionality which will be described in 4.4 User Interface and Documentation.

In 4.3 Implementation I describe the implementation which was part of this thesis. This is mostly the integration code and some changes which were necessary for a tool to work properly inside EiffelStudio. Also the two additional features which were implemented are shortly described.

There also are a screenshot and additional diagrams at the end of this document in the appendix.

The project is hosted on Origo [8] and can be accessed via this project page: http://ebbro.origo.ethz.ch/.

For additional documentation or people who are interested in the source code, I would recommend visiting the project page.

4.2 Previous work
The Ebbro project started in the year 2007 as an Eiffel Lab Project at ETH. This project was done by me and was also supervised by Marco Piccioni. The results of this first phase of the project was a standalone GUI application to browse through serialized objects, which were stored via the Eiffel independent store mechanism. Additionally a filter mechanism for attributes inside an object was implemented.

In the year 2008 Pascal Dufour again started an Eiffel Lab Project whose target was to add a write-back mechanism to Ebbro. This means that simple values could be edited inside the tool and then be serialized back. This was also only for the binary format. This lab project met its result, but there were still some problems with correctly serializing the objects back and with editing values inside structures. The binary format is tricky to correctly reconstruct with the right header information, which is needed so that the original system would not reject the object file. Also editing inside structures (tuples, specials, hashables…) did not work.

These problems have been solved by the implementation described in this thesis.
4.2.1 Ebbro inside EiffelStudio

A small diagram which should illustrate the Ebbro tool inside EiffelStudio:

Figure 3 Ebbro tool overview

The ebbro_tool cluster is located here: Src → Eiffel → interface → new_graphical.

The ES_EBBRO_CONTROLLER class is the glue code between the serialize/deserialize classes (located in the writer/reader cluster) and the GUI. The main GUI class is ES_EBBRO_TOOL_PANEL, which takes the role of the MAIN_WINDOW in the standalone version of Ebbro. So in the main cluster of the Ebbro tool are all the GUI specific classes (like Dialogs, Grids, Filter…) and also the classes that hold the constants used in Ebbro (such as ES_EBBRO_NAMES…).

The other main part of Ebbro is the dADL library and located here: Src → library → dadl

The complete source code can be accessed on the Eve SVN directory: http://svn.origo.ethz.ch/wsvn/eiffelstudio/branches/eth/eve.
4.2.2 Main Classes

**Ebbro Tool**
- *ES_EBBRO_TOOL*: The Ebbro tool class.
- *ES_EBBRO_TOOL_PANEL*: The main class of the Ebbro tool. Holds GUI information/layout.
- *ES_EBBRO_CONTROLLER*: The controller in the model-view-controller pattern. The glue between deserializer/serializer and the GUI.
- *ES_EBBRO_GRID*: The tree view to browse through the objects.
- *EB_EBBRO_TOOL_DATA*: The preference class for all the GUI preferences.
- *ES_EBBRO_NAMES*: Holds the GUI strings.
- *ES_EBBRO_TOOL_ICONS*: Ebbro's icons class.

**EiffelStudio Integration**
- *EB_DEVELOPMENT_WINDOW_MENU_BUILDER*: Add the tool to the menu.
- *INTERFACE_NAMES*: All GUI names in the system.
- *EB_DEVELOPMENT_WINDOW_TOOLS*: Also add the tool here.
- *EB_GUI_PREFERENCES*: In case you want to add preferences...

**EiffelStudio Helpers**
- *EB_SHARED_WINDOW_MANAGER*: Gives access to the last focused development window.
- *ES_DOCKABLE_TOOL_PANEL*: *ES_EBBRO_TOOL_PANEL* inherits from this class, as it provides all functionality for a dockable tool panel.
- *EB_SHARED_PREFERENCES*: To access shared preferences.
- *EB_FILE_OPEN_DIALOG*: An EiffelStudio file open dialog, which remembers the last folder.
- *EB_CHOOSE_CLASS_DIALOG*: An EiffelStudio dialog to choose a class from the system.

4.2.3 Integration

Since the main goal of the thesis in this part is to integrate the standalone tool Ebbro into Eve, there are no new concepts or patterns used in this part of the implementation.

The integration was pretty straightforward, once I figured out where to insert the Ebbro classes and what changes had to be made.

The icon handling had to be changed and adapted to the EiffelStudio way of importing/handling icons. The way this was implemented is documented here more or less: [http://dev.eiffel.com/Icon_Resources](http://dev.eiffel.com/Icon_Resources).
Of course all dialogs and prompts were reconfigured to use the built in EiffelStudio mechanisms (e.g. EB_FILE_OPEN_DIALOG…).

4.2.4 Object Compare Feature

This feature was not on the project plan for this master thesis, but in my opinion it really helps developers and makes Ebbro more attractive.

The implementation for comparing the attributes of two objects is the following:

A list of all the attributes of the first object gets traversed in a breadth-first manner. The fields get compared to the second list (which holds the attributes of the second object). By highlighting the differences and marking every attribute in the second list it works its way forward. At the end of this first list traversal all the attributes in the second list, which are not marked, are simply highlighted as being “additional” in the second object. The attributes are being compared based on their names and then the value and type have to match.

4.2.5 Custom Serialization Feature

This feature, not part of the original project plan for this thesis, was added to enhance Ebbro.

This feature helps a developer to create a custom serialized form. Especially for larger objects with a lot of attributes this feature can be useful. One can easily select the attributes and then the code gets automatically generated.

It basically generates the code used to programmatically specify which attributes to store and the developer can copy/paste this code.

This feature needs three different GUI parts which were implemented:

1. The EB_CHOOSE_CLASS_DIALOG is used to let the user select the class. And the pick and drop mechanism for the feature’s toolbar icon is implemented.

2. A custom dialog class ES_EBBRO_CUSTOM.Serialization_DIALOG which enables the user to select the attributes. Inside the dialog an EV.CHECKABLE_LIST filled with all the attributes of the class is used. The attributes of the class are retrieved by the CLASS_C object and then through the COMPUTED_FEATURE_TABLE class.

3. A custom output dialog, which simply outputs the result of the automatic code generation and provides a copy/paste functionality to the user.

The real meat of this feature (but not the hardest part to implement) is the automatic code generation. This is pretty straightforward and is done by the class ES_EBBRO_CODE_GENERATOR and all the constants used are in the ES_EBBRO_CODE_GENERATOR_CONSTANTS class.
4.3 User Interface and Documentation

4.3.1 Getting Started

Probably watching the how-to video here is a good start: [http://ebbro.origo.ethz.ch/download](http://ebbro.origo.ethz.ch/download) (Ebbro (integrated) Demo). Or you could peek at the screenshot in the appendix of this document.

To get started you should activate Ebbro inside EiffelStudio here: View → Tools → Ebbro.

Figure 4 View Ebbro inside ES
4.3.2 Main Features

**Toolbar**
This section explains the toolbar and its buttons:

![Ebbro Toolbar]

From left to right:
1. Open an object file
2. Custom serialization feature
3. Compare two objects
4. Move selected root object to the left of split screen
5. Move selected root object to the right of split screen
6. Undo edit operation on selected root object
7. Redo edit operation on selected root object
8. Remove selected root object from the browser-view
9. View menu

**Opening an Object**
You can open an object by clicking the open file image in the toolbar.

The corresponding object will be opened and will get the name "Root Object #" (the tool tip will display the objects location in your file system).

**Supported Objects**
1. Serialized objects which were stored via the Eiffel Independentent Store mechanism (with or without fast retrieval flag)
2. Serialized objects in the dADL format

Note: Ebbro supports more than one object inside the object browser. Simply open another object after your first one...
**Root Object Features**

Use this functionality by right clicking on a root object. This menu will appear:

![Ebbro Right Click Menu](image)

1. Remove: Will remove the object from the browser.
2. Move: If the split screen is enabled, you can simply move the object from one side to the other.
3. Filters: Filter the root object - for details see the corresponding section.
4. Save: Save the root object back to the file - in the same format it was deserialized.
5. Save as...: Save the root object back to a file - for details see the corresponding section.

Note: You can also remove an object from the browser by selecting the root object and pressing 'Del' on your keyboard or by using the toolbar button.

**Object Pixmaps**

Objects have a pixmap corresponding to their type:

- ![Reference type](image) Reference type (user created or library type)
- ![Character or string type](image)
- ![Numeric type](image)
- ![Boolean type](image)
- ![Void reference](image)
- ![Container type](image)
- ![Tuple type](image)
- ![Pointer type](image)
**View Menu**

Figure 7  Ebbro View Menu

- Enable Cyclic Browsing: Enable/disable the possibility to browse cyclic objects.
- Show Address Column: Shows/hides the address column.
- Split Screen: Use split screen or not.
- Update Addresses: Updates all the physical object addresses in the address column.
- Filter Options: Select your preferred filter option

### 4.3.3 Advanced Features

**Filtering on Root Objects**

First you can choose whether you want to filter in or out on your root objects: by selecting your choice in Figure 7.

- Filtering in means that *only* the objects which apply to the given filter will be shown.
- Filtering out means that the objects which apply to the given filter will *not* be shown.

Now you can choose between three filters:

Figure 8  Ebbro Filters

1. None: trivial filter which does not filter anything.
2. Void: to filter void objects.
3. Cycle: to filter cyclic objects.
Note: Select any root object and in the top right corner you will see the current filter which is applied to the selected object:

Figure 9  Ebbro Filter Text

---

**Cyclic Object Highlighting**

By trying to browse a cyclic object (and by default cyclic browsing is disabled) you will be noticed by an object pixmap-change (◯) and additionally the identical physical object addresses will be highlighted.

**Updating Objects**

Once an object has been loaded, the primitive types can be changed by clicking on them:

Currently, the system supports editing of the following types:

1. String
2. Character
3. Integer
4. Natural
5. Boolean
6. Real and Double

**Writing Objects back**

After editing, the object can be written back to a file. In order to do this, right-click on the object root, then select 'Save as...' or 'Save'.

Two cases:

1. Save: In case you select save, the object will be written back to the file from which it was opened and in the same format.
2. Save as: In this case you can select a file and also choose the format (by selecting a filter).

In both cases: If the file doesn't exist, it will be created, and otherwise it will be overwritten with the object that is serialized.
4.3.4 Custom Serialization Feature

The steps are:

1. Choose the class you want to create a custom serialized form for
   - By drag/drop a class onto the toolbar button
   - By selecting the class in the dialog, which appears when clicking on the toolbar button

2. Choose the attributes you want to serialize. By clicking on them and/or using the buttons: Select – All / None / Invert.

3. Copy/Paste the generated code.

Code Sample

This is a source code sample of a developer who used the custom serialization feature and has copied/pasted the generated code:

```eiffel
custom_serialize_example is
  -- serialize an object with a custom serialized form
  local
    manager: DADL_PERSISTENCE_MANAGER
    custom_form: SERIALIZED_FORM
    object: A_CLIENT
  do
    -- create object to store
    create object
    create custom_form object
    create custom_form.make
    -- add attributes through the generated code
    create_custom_form_for_a_client(custom_form)
    -- create persistence_manager and add our serialized_form
    create manager.make ("c:/a_client.adls")
    manager.format.set_serialized_form(custom_form)
    -- serialize 'a_client' to the filesystem
    manager.store (object)
  end

create_custom_form_for_a_client (a_form: SERIALIZED_FORM) is
  -- Adds the custom serialization to 'a_form' object
  require
    not_void: a_form /= void
  local
    l_list: ARRAYED_LIST[STRING]
  do
    create l_list.make (1, 3)
    l_list.extend ("my_string")
    l_list.extend ("my_string2")
    l_list.extend ("my_int")
    a_form.put_serialized_form(l_list, "A_CLIENT")
  end
```

26
4.3.5 Object Compare Feature
This feature will compare to objects and highlight possible differences. The feature is format independent! To use the tool, click on the toolbar icon and then select two object files.

Color encoding:
- The two objects are equal.
- This object structure contains a difference.
- Actual attribute which is different.
- An additional attribute in the left object.
- An additional attribute in the right object.

4.3.6 Preferences
Ebbro also stores some preferences, which are accessible under Tools → Preferences:

Figure 10  Ebbro Preferences

Short Explanation (from top to bottom):
1. Cyclic browsing: If enabled, you can browse through cyclic object structures and you won’t get a hint. (View Menu)
2. Filter in: Filter in enabled/disabled. (View Menu)
3. Show addr column: Address column visible or not. (View Menu)
4. Show overwrite question: Show overwrite question when selecting ‘Save’ on a root object. (Only editable here under Preferences)
5. Split position: The split screen position.
6. Split screen: Split screen is shown or not. (View Menu)
5 Challenges

5.1 Additional Work

Almost every project has some parts needing to be implemented or solved that were not anticipated when writing the project plan. The biggest efforts were:

- In the existing dADL project implementation the whole mechanism, which was already implemented, is based on recognizing the type information. So the INTERNAL class of Eiffel can be used to inspect the objects. But in order to use the dADL library for a tool like Ebbro, which should be able to serialize/deserialize objects from any system and mostly of course objects, whose types are not known to the system. So I built another serialization mechanism, which does not use the help of the class INTERNAL. This means when deserializing an object from the dADL file, all the helpful features from the INTERNAL class cannot be used (e.g. is_special_type, generic_type_of_count, field_count…) because this class only works on known types. Therefore the implementation has to gather the needed information elsewhere (e.g. analyzing class names, dt_structure of objects…).

- The previous lab work of Ebbro didn’t correctly serialize the header information in the binary format. This needed to be fixed, in order that the original system, which serialized the object, would recognize and not reject an object file, which was created by Ebbro. It was quite a challenge to fix the implementation, so that it would correctly serialize the header.

- After the previous lab work of Ebbro the user wasn’t able to edit primitive fields inside structures (e.g. inside a list, hashtable, tuple…). This is of course not very user-friendly and had to be fixed. To add the functionality to edit inside all these structures was a huge challenge and took a lot of time to make it work properly. The edit operations were also not propagated to all other possibly shared objects. So I added the feature, that all edit operations would be propagated throughout the GUI to all shared objects.

I also added additional features to the Ebbro tool which were not originally planned. Namely:

1. Custom Serialized Form Feature
2. Object Compare Feature
3. Undo / Redo possibility for edit operations

5.2 Difficulties

- Understanding how the dADL parser is generated by the gobo binaries and how to extend the parser to recognize new patterns.

- Finding an implementation, which respects the dADL specification for shared objects in the serialize/deserialize methods. Afterwards testing and making sure all aspects are taken into account.
• Writing the correct header information for binary objects inside Ebbro.
6 Issues

6.1 dADL Library
All issues are being reported on the project site on Origo: http://dadle.origo.ethz.ch/issues.
Main issues were:

- We had quite a discussion on how to implement the custom serialized form. There are different ways to implement and to design this. One approach could be the use of inheritance to specify which attributes to serialize. This would result in inheriting from a serialized form class and then redefine a list holding all the attribute names. Another approach is using a client relationship, but also in this case the persistence code would be coupled with the class code (via an attribute). We then decided to use a ‘persistence-agnostic’ approach and to have a $\texttt{SERIALIZED\_FORM}$ class in the framework containing the information for all possible serialized forms. So the $\texttt{PERSISTENCE\_FORMAT}$ class now has an attribute ‘serialized_form’ and from there every format (e.g. $\texttt{DADL\_FORMAT}$) can retrieve the corresponding list with the attribute names to be serialized.

- We also had an issue about storing or not storing void attributes. The existing dADL project was not storing void attributes. But for example list types and other container types with ‘gaps’ in them, i.e. void members, would not work properly. We decided to also serialize void references and choose the syntax <void>.

- Still an open issue is about expanded types. Expanded objects are currently not correctly serialized and deserialized. This is mostly because of missing or incorrect handling of expanded objects in the $\texttt{INTERNAL}$ class.

6.2 Ebbro
All issues are being reported on the project site on Origo: http://ebbro.origo.ethz.ch/issues.
Main issues were:

- There were issues about implementation difficulties and also about the fact that Ebbro produced wrong header information for binary objects. But these issues could be solved in the end.

- A big problem in Eiffel 6.3 is a bug in the $\texttt{INTERNAL\_HELPER}$ class, which checks the syntax of a class name. The bug is that class names like $\texttt{A\_CLIENT}$, which have an underline at the second position, get rejected although this class name is valid. This bug should be fixed in version 6.4.
7 Conclusions

7.1 Conclusions
In this thesis I have implemented a new serialization library for Eiffel, which uses the dADL format. This library is now already being used inside the Ebroo tool as a result of this master thesis.

The Ebroo tool is now integrated into EVE and ready to be used by eager Eiffel developers. The tool will be a big help for developers using object persistence in their applications. The custom serialized form feature can even make life a bit easier for developers who want to specify such a custom serialized form. Ebroo is probably also really useful for testing purposes. The developer can easily open the serialized objects and quickly browse through the attributes or can even use the compare feature, to compare against a predefined result.

I really enjoyed working on this master thesis. I had a lot of different aspects to cover and to implement. On one hand a new text-based serialize format to look into and to build a library with this new format. On the other hand working on a GUI application and adding a new tool to a large IDE with lots of interesting code to look at.

I didn’t get bored because there were always new challenges awaiting me and even having the time to come up with additional features to implement was great. In my opinion these ‘unplanned’ features really improve or complement the overall result of this thesis.

7.2 Future Work
In the dADL library one will have to look at the serializing/deserializing of expanded types once the INTERNAL class provides correct handling of expanded types.

Possible future Ebroo development steps are:

- Also allowing updating references (not just simple value types).
- Assigning labels to attributes; e.g. to make cyclic object highlighting more convenient for the user.
- Extend the functionality of the object compare feature to directly compare two objects, which are already opened in the browser.

Once there is a new serialization format in Eiffel, this should be added to Ebroo by adding a new deserializer/serializer for this format.
8 References and Background Material

8.1 Reading list

Chair of Software Engineering: Semester-/Diplomarbeiten; Online at:
http://se.inf.ethz.ch/projects/index.html


ADL format documentation:

The persistence framework:
http://dev.eiffel.com/Persistence_unified

8.2 References

[1] ADL format:


[7] DT_OBJECT_CONVERTER class:


[11] Eiffel Programming Language:

9 Glossary

Object: In computer science, an object commonly means a data structure consisting of data fields (attributes) and routines (or methods) that can manipulate those fields. (src: Wikipedia.org)

Object file: In this document the term object file is used for a file, which content is the serialized version of an object. (Not the source code of the implementation)

Root Object: An object which was opened from the file system (denoted with Root Object #)

Cyclic Object: An object which will lead to a cycle by browsing it. For example object A references an object B and object B also references object A → by expanding this attribute which references object B inside the object A: one ends up in a cycle.

Shared Object: An object which is referenced by more than one other object and is by this somehow shared between its clients.

GUI: Graphical User Interface

IDE: Integrated Development Environment

SVN: Subversion (SVN) is a version control system initiated in 2000 by CollabNet Inc. It is used to maintain current and historical versions of files such as source code, web pages, and documentation. Its goal is to be a mostly-compatible successor to the widely used Concurrent Versions System (CVS). (src: Wikipedia.org)
Appendix

A 1 dADL Library

Example Usage

```
feature -- serialize

serialize (a_path: STRING;an_obj: ANY) is
  -- serialize object to a file_path without type information
  do
    create Persistence_manager.make(a_path)
    Persistence_manager.serialize_type_information(true)
    Persistence_manager.store (an_obj)
  end

serialize_without_type_information (a_path: STRING;an_obj: ANY) is
  -- serialize object to a file_path without type information
  do
    create Persistence_manager.make(a_path)
    Persistence_manager.serialize_type_information(false)
    Persistence_manager.store (an_obj)
  end

feature -- deserialize

deserialize (a_path: STRING): ANY is
  -- deserialize dt structure given a file_path
  do
    create Persistence_manager.make(a_path)
    result := Persistence_manager.retrieve
  end

deserialize_non_typed_dt (a_path: STRING;a_type_id: INTEGER): ANY is
  -- deserialize without type information in the DT structure
  -- 'a_type_id' represents the class_type e.g. from internal
  -- 'dynamic_type_from_string'
  do
    create Persistence_manager.make(a_path)
    Persistence_manager.set_type_id(a_type_id)
    result := Persistence_manager.retrieve
  end

feature -- access

  Persistence_manager: DADL_PERSISTENCE_MANAGER
```
### Performance Test Results

The first table shows the results for serializing and the second one for deserializing.

<table>
<thead>
<tr>
<th># Attributes</th>
<th>STRING</th>
<th>CUSTOM OBJECT</th>
<th>INTEGER</th>
<th>SHARED OBJECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0.0159</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.016</td>
<td>0.0159</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0.1859</td>
<td>0.0149</td>
</tr>
<tr>
<td>5000</td>
<td>0.016</td>
<td>0.0159</td>
<td>3.328</td>
<td>0.016</td>
</tr>
<tr>
<td>Avg</td>
<td>0.004</td>
<td>0.033975</td>
<td>0.69046</td>
<td>0.0157</td>
</tr>
<tr>
<td>binary/sADL</td>
<td>224</td>
<td>316</td>
<td>128</td>
<td>222</td>
</tr>
<tr>
<td>C/sADL</td>
<td>462</td>
<td>703</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Attributes</th>
<th>STRING</th>
<th>CUSTOM OBJECT</th>
<th>INTEGER</th>
<th>SHARED OBJECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0.2339</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.0619</td>
<td>0.25</td>
</tr>
<tr>
<td>1000</td>
<td>0.016</td>
<td>0.016</td>
<td>0.1859</td>
<td>0.5</td>
</tr>
<tr>
<td>5000</td>
<td>0.1569</td>
<td>0.0309</td>
<td>3.844</td>
<td>0.3119</td>
</tr>
<tr>
<td>Avg</td>
<td>0.042975</td>
<td>0.011725</td>
<td>1.0232</td>
<td>0.32369</td>
</tr>
<tr>
<td>binary/sADL</td>
<td>87</td>
<td>164</td>
<td>1.46</td>
<td>132</td>
</tr>
<tr>
<td>C/sADL</td>
<td>23</td>
<td>38</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Overview Bon-Diagram
A 2 Ebbro

Screenshot

Overview Bon-Diagram