Master Thesis

Reflection for Eiffel

Author(s):
Fluri, Beat

Publication Date:
2004

Permanent Link:
https://doi.org/10.3929/ethz-a-005115700

Rights / License:
In Copyright - Non-Commercial Use Permitted
Reflection for Eiffel

Master Thesis

Beat Fluri

Supervised by
Prof. Dr. Bertrand Meyer
Bernd Schoeller

Chair of Software Engineering
Department of Computer Science
ETH Zürich

September 2004
Master thesis

Degree: Master of Science ETH in Computer Science

Specialization: Software Engineering

Author: Beat Fluri, fluri@computerscience.ch

URL: fluri.computerscience.ch/projects/

Project period: Monday, March 15, 2004 - Wednesday, September 15, 2004

Chair of Software Engineering

Department of Computer Science, ETH Zürich
Acknowledgments

I would like to thank the following people for various reasons:

- My parents, for making my studies on computer science possible and backing me in every situation.
- My lovely fiancée Janine Koller, for giving me the courage to keep following my intentions and dreams and for being the main reason to live every day.
- My best friend Philipp Traber, for supporting me in making my decisions, for always listening to me and for always being there, if I need him.
- Bernd Schoeller, for help and support on this master thesis.
- Bertrand Meyer, for giving me the opportunity to make this master thesis.
- Petra Marty, for her very helpful corrections and suggestions for improving the understandability of my master thesis.
- Christopher Nenning and specially Olivier Jeger, for their constructive comments and support on my work.
- My friends from ETH: Rolf Bruderer, Olivier Jeger, Christoph Kiefer, Petra Marty, Christopher Nenning and Roger Rüegg, for having such a great time.
- Hans Dubach, for his outstanding help on administrative questions related to my studies at ETH.

Zürich, in September 2004

Beat Fluri
Abstract

Reflection is the ability to discover object properties at runtime. We use reflection in various areas of Software Engineering nowadays. Its scope is beyond a simple debugger. Examples are object persistence, monitoring and automatic configuration. Eiffel does not have a reflection library yet.

In this thesis we evaluate reflection APIs of various state of the art programming languages and runtime environments. We point out that we are not fully satisfied with their designs. Based on these investigations we design a reflection library for Eiffel. We suggest the approach to map the Eiffel object model as exhaustively as possible to a simple interface. All of the evaluated programming languages, except Smalltalk, do not care for such an approach.

As an example application of the library we built the ObjectBrowser to browse a given object structure. We are able to modify the values of the objects’ basic type attributes and to visualize the object structure in a graph view.
Zusammenfassung


Als Beispiel-Anwendung der Bibliothek entwickelten wir den ObjectBrowser, um eine gegebene Objekt-Struktur zu durchstöbern. Wir können die Werte der Basis-Typ Attribute verändern und die Objekt-Struktur in einer Graph-Ansicht anzeigen lassen.
Contents

1 Introduction

2 Existing Reflection APIs
  2.1 Introduction
  2.2 The Java reflection API
    2.2.1 Reflection in Java
    2.2.2 Class design
    2.2.3 Example use of the API
    2.2.4 Forthcoming Java release
    2.2.5 Evaluation of the Java reflection API
  2.3 The .NET Framework reflection API
    2.3.1 Class design
    2.3.2 Example use of the API
    2.3.3 Forthcoming .NET Framework release
    2.3.4 Evaluation of the .NET reflection API
  2.4 Reflection in Smalltalk-80
    2.4.1 A short introduction to Smalltalk
    2.4.2 Reflection facilities
    2.4.3 Evaluation of the reflection facilities of Smalltalk
  2.5 Reflection in scripting languages
    2.5.1 Ruby as an example
  2.6 Existing reflection in Eiffel
    2.6.1 Describing agent expressions
    2.6.2 Kernel Library class INTERNAL
  2.7 Conclusion
    2.7.1 Expected set of reflection facilities
    2.7.2 Expected design of a reflection API

3 Reflection for Eiffel
  3.1 Introduction
  3.2 Requirements
    3.2.1 Minimal set of reflection facilities
  3.3 Design of ERL
    3.3.1 Separation of type and class
    3.3.2 Type
    3.3.3 Class
    3.3.4 Features in Eiffel
    3.3.5 Features in ERL
    3.3.6 Routines

4 Expected Design of a Reflection API

5 Conclusion
### Contents

| 3.3.7 Queries | 39 |
| 3.3.8 Access to ERL | 39 |
| 3.4 Partial implementation of ERL | 40 |
| 3.4.1 Universe | 41 |
| 3.4.2 Type | 41 |
| 3.4.3 Class | 44 |
| 3.4.4 Features | 45 |
| 3.5 Conclusion | 47 |
| 4 ObjectBrowser | 51 |
| 4.1 Introduction | 51 |
| 4.2 Requirements and functionality | 51 |
| 4.3 Implementation | 54 |
| 4.3.1 Implementation of the GUI | 54 |
| 4.3.2 Object structure layout | 55 |
| 4.3.3 Platform independence | 57 |
| 4.4 User guide | 58 |
| 4.4.1 Requirements | 58 |
| 4.4.2 Installation | 58 |
| 4.4.3 How to use ObjectBrowser | 59 |
| 5 Conclusion | 63 |
| 5.1 Summary | 63 |
| 5.2 Main contribution | 63 |
| 5.3 Future work | 64 |
| A ERL API | 65 |
| A.1 ERL_ARGUMENT | 65 |
| A.2 ERL_ATTRIBUTE | 66 |
| A.3 ERL_CLASS | 67 |
| A.4 ERL_CLUSTERS | 68 |
| A.5 ERL_CONSTRAINT | 69 |
| A.6 ERL_FEATURE | 70 |
| A.7 ERL_FEATURE_CLAUSE | 73 |
| A.8 ERL_FUNCTION | 74 |
| A.9 ERL_GENERICS | 75 |
| A.10 ERL_PROCEDURE | 76 |
| A.11 ERL_QUERY | 77 |
| A.12 ERL_ROUTINE | 78 |
| A.13 ERL_TYPE | 80 |
| A.14 ERL_UNIVERSE | 83 |
| B Scope of the ERL implementation | 85 |
| References | 89 |
List of Figures

Figure 2.1: Class diagram of the most important classes ................................................. 4
Figure 2.2: New reflection on genericity interfaces in Java 1.5 ........................................... 10
Figure 2.3: A better class/interface relation ........................................................................ 13
Figure 2.4: Inheritance hierarchy of System.Type .............................................................. 14
Figure 2.5: Constructor, Method relation in .NET .............................................................. 15
Figure 2.6: Class/meta class kernel ..................................................................................... 19
Figure 3.1: Main parts of ERL ............................................................................................. 28
Figure 3.2: Example hierarchy to show lookup binding ..................................................... 31
Figure 3.3: Feature categories as described in Mey97 ......................................................... 35
Figure 3.4: Feature categories ............................................................................................. 35
Figure 3.5: Features reflection in ERL .................................................................................. 36
Figure 3.6: Example of merging features ............................................................................ 36
Figure 3.7: Bridge pattern: implementation by inheritance ................................................. 40
Figure 3.8: ObjectBrowser ................................................................................................. 53
Figure 4.1: Graph view of object structure from figure 3.7 ................................................. 53
Figure 4.2: A simple graph lay-outed with dot .................................................................. 55
Figure 4.3: The ObjectBrowser main window ................................................................... 60
Figure 4.4: Kind of nodes in the object tree ........................................................................ 60
Figure 4.5: Declared type vs. attached type of an attribute .............................................. 61
Figure 4.6: The ObjectBrowser’s toolbar ........................................................................... 62
Figure 4.7: Graph layout of object structure from figure 4.3 with depth 4............................. 62

List of Tables

Table 2.1: Comparison of reflection facilities ...................................................................... 23
Table 3.1: Minimal set of reflection facilities ....................................................................... 26
Table 3.1: (continued) ........................................................................................................ 27
Table 3.7: The reasons why we have only implemented some facilities partially ................. 48
Table 4.1: Options of GRAPH_VISUALIZER .................................................................... 56
Chapter 1

Introduction

We use the ability to discover object properties at runtime more and more frequently. There are two kinds of object properties. First, we know nothing about the object and want to get to discover the interface of it. Second, we know the object, but in some circumstances we need some information unavailable to normal clients, e.g. private fields.

Most of the state of the art programming languages and runtime environments provide such facilities. Normally, we speak about reflection or introspection to summaries them. To stay consistent, we only use the term reflection in this thesis.

The first two obvious application areas of reflection are debugging and object persistence. There is no way to get or set the values of private fields directly, therefore we must get around these limits to make better propositions of a system or to store the state of an object. Reflection also helps to improve the engineering trade-offs readability, performance and code size.

In fact, Eiffel does not provide a complete reflection library yet. There exists only an Eiffel class INTERNAL to get information about the fields of an object and to modify them.

Various reflection APIs of well known programming languages and runtime environments exist. We will evaluate some of them to get an idea how we should design a good reflection library and what facilities it should provide.

Designing a reflection library is certainly an old field of research, but our approach to map a very sophisticated object model into a reflection library is new in this field. The implementation of the Eiffel Reflection Library (ERL) is not feasible to full extend, because the compiler and runtime environment of Eiffel does not support more reflection facilities than available in the class INTERNAL.

To show the usage of the reflection library, we will build the ObjectBrowser application. We will be able to traverse object structure with it and modify their fields.

A short industrial example

The Apache Jakarta Tomcat, a servlet container [TASF], uses reflection to start and configure itself. Tomcat is used as the reference implementation of the Java Servlet API and JavaServer Pages technology.

The server is configured using XML files. Tomcat then uses Reflection to configure the server from these files. For that an XML parser parses the configuration file. At the beginning it contains the line:

<Server port="8005" shutdown="SHUTDOWN">

The parser maps the tag name Server to the class name StandardServer and creates a new instance of it. After that it configures the server with the two attributes. For that, it looks up a
set method for `port` taking an `int` parameter and one for `shutdown` taking a `String` parameter. If it finds the desired methods it invokes them with the given attributes’ value also via reflection.

The whole Tomcat server is started and configured in this way.
Chapter 2

Existing Reflection APIs

2.1 Introduction

Most common programming languages provide reflection. The Application Programming Interfaces (API) of the reflection libraries can be very different. For example: a constructor is not a method in Java, but in .NET it is, or: only Smalltalk and Ruby separate class and instance variables.

We discuss the following popular programming languages in this chapter (SDK stands for Software Development Kit):

- **Java 2 SDK 1.4.2.** Java is widely used nowadays - maybe the most used programming language. Many tutorials and examples for reflection in Java exist.

- **Java 2 SDK 1.5 beta.** Sun is adding genericity to the forthcoming version of Java, but it will be an extension of the language and not of the type system (Java Virtual Machine, JVM). A beta release is already available. It is interesting to see, how a well-known programming language deals with genericity and its reflection.

- **.NET Framework SDK 1.1.** The .NET Framework from Microsoft is the latest known runtime system. It has many improvements over Java. We can expect that they also enhanced the reflection API. The examples we will show are all written in the .NET Framework language C#.

- **.NET Framework SDK 2.0.** Microsoft is also adding genericity to the .NET Framework (v2.0) and it will be an extension to the Common Language Runtime (CLR) and not only to the language C#. This issue is different as in Java 1.5 and is, as we will see, very important in terms of reflection.

- **Smalltalk-80 in Squeak v3.6.** Reflection is older as expected. Smalltalk uses reflection to work. Therefore, it is rather a part of the systems as an API. Since Smalltalk is one of the oldest object-oriented languages, it is very interesting to see how it is implemented and how Smalltalk used reflection in itself.

- **Ruby 1.8.1.** The interpreters of scripting languages use mostly reflection. We have chosen Ruby as a reference of reflection facilities in scripting languages. Ruby is a new and object-oriented language.

- **ISE EiffelStudio 5.4.** Eiffel does not miss reflection completely. It already provides partial reflection facilities. We will look at them shortly.
We start with the discussion about the Java Reflection API, because we think that Java reflects popular opinions of object-oriented programming and its corresponding reflection. We will give an introduction to reflection in general along with the discussion of the reflection API of Java.

In the remaining sections we compare Java with the other languages introduced in the list above.

This chapter concludes with a feature comparison of the languages for giving an idea of what a reflection library should provide.

### 2.2 The Java reflection API

#### 2.2.1 Reflection in Java

Reflection means to discover properties of objects at runtime. Java uses the following list of object properties:

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object’s field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array’s components.

We do not discuss every item of the list above.

#### 2.2.2 Class design

We can find all except one of the involved classes for reflection in the package `java.lang.reflect`; the class `Class` is in the package `java.lang`. Nevertheless, it provides the main access to the reflection of Java.

![Class diagram of the most important classes](image.png)
2.2 The Java reflection API

Figure 2.1 shows the class diagram of the most important classes of the package java.lang.reflect. The list below describes all classes related to reflection in Java.

- **java.lang:**
  - **Class.** Main access to reflection. We can get constructors, fields and methods. We can retrieve these members using names, or signatures in case of methods.

- **java.lang.reflect:**
  - **Member.** Interface to members. Provides information about the declaring class, the modifiers and the name.
  - **AccessibleObject.** Object accessibility abstraction during runtime.
  - **Array.** Setters and getters of the values of an array.
  - **Constructor.** Information about the formal arguments. Instantiating an object of the constructor’s reflected class.
  - **Field.** Setters and getters of the values of a field.
  - **Method.** Information about the formal arguments and the return type. Method invocation.
  - **Modifier.** Constants describing the modifiers of members, like abstract, final, static, etc.
  - **Proxy.** Possibility to interpose some behavior before and after a method invocation at runtime. Proxies will not be explained in this section. For further information read [Har01].

We have two possibilities to obtain an instance of a particular class:

1. Static: `String.class` to get the class description of the type `String`.
2. Dynamic: `aString.getClass()` to get the class description of the type of the object attached to `aString`.

2.2.3 Example use of the API

We do not only use reflection because it is necessary. It can also help to keep code more readable. Next we look at two examples to describe these scenarios. For the sake of simplicity exception handling is omitted in the sample codes.

**Necessity of reflection**

It is almost impossible to implement persistence of objects without the use of reflection. The main reason are the private fields. We cannot assume that developers implement getters and setters for each private field. If they do not provide them, we are not able to get and set the values of private fields. Nevertheless we must have a possibility to access them and the solution is reflection.
Consider the following simple class:

```java
package reflection.examples;

public class Person {

    private String name = null;
    private int age = 0;

    public Person (String name) {
        this.name = name;
    }

    public void setAge (int age) throws IllegalArgumentException {
        if (age < 0 || age > 150) {
            throw new IllegalArgumentException("age must be in (0, 150) nonatomic");
        }
        this.age = age;
    }

    public void getAge () {
        return age;
    }

}
```

One possible solution to store an instance of this class using reflection might be:

```java
Person person = new Person("John");
person.setAge(18);

Class claz = p.getClass();
String className = claz.getName();

// store 'className' in database

Field[] fields = claz.getDeclaredFields();
String fieldName = null;
Object value = null;
for (int i = 0; i < fields.length; i++) {
    fieldName = fields[i].getName();
    value = fields[i].get(person);

    // store ('fieldName', 'value') in database
}
```
In this example an exception occurs during runtime, because we cannot access private fields without making them accessible with `setAccessible(true)`:

```java
...
for (int i = 0; i < fields.length; i++) {
    fields[i].setAccessible(true);
    ...
}
```

The same semantic applies also to constructor and methods.

Another point to mention is the difference between the methods

```java
public Field[] getDeclaredFields ()
```

and

```java
public Field[] getFields ()
```

The first one returns all declared fields, even the non-public, of the class without considering inherited fields. The second one returns all public fields of the class and of its ancestors. The same applies for methods and constructors.

**Reflection helps on code readability**

As mentioned above, we also use reflection to simplify the code.

Assume a `turtle` application takes an infinite number of command line arguments to move the turtle to draw a line. These arguments are pairs of a direction (`up`, `down`, `left` or `right`) and a number of steps. The command line argument parser of this application may look like:

```java
public static void main (String[] args) {

    Turtle turtle = new Turtle ();

    for (int i = 0; i < args.length; i += 2) {
        if (args[i].equals("up")) {
            turtle.up (Integer.parseInt (args[i+1]));
        }
        else if (args[i].equals("down")) {
            turtle.down (Integer.parseInt (args[i+1]));
        }
        else if (args[i].equals("left")) {
            turtle.left (Integer.parseInt (args[i+1]));
        }
        else if (args[i].equals("right")) {
            turtle.right (Integer.parseInt (args[i+1]));
        }
    }
}
```
The use of reflection makes the code much simpler:

```java
public static void main(String[] args) {

    Turtle turtle = new Turtle();
    Class claz = turtle.getClass();

    Class[] params = { int.class };  
    for (int i = 0; i < args.length; i += 2) {
        Method method = claz.getMethod(args[i], params);
        Object[] arguments = { new Integer(args[i+1]) };
        method.invoke(turtle, arguments);
    }
}
```

Because of the following reasons, this sample code is slower than the one without reflection in respect of performance:

- The method lookup `claz.getMethod(args[i], params)` is slower than evaluating an if-condition.
- We must create an extra object array for the argument.
- Invoking the method via reflection has an additional overhead.

If we cached the method table, the method lookup would perform faster.
2.2 The Java reflection API

2.2.4 Forthcoming Java release

Regarding the Eiffel Reflection Library, we must have a look at Java 1.5, because it introduces genericity and the corresponding reflection facilities.

Genericity

The syntax of a generic declaration is familiar to the one of C++ templates. For example, a generic class declaration:

```java
public class List<T>
```

or a generic method declaration:

```java
public <T> void printAll(List<T> list)
```

Such as Eiffel, Java has constraint and multiple constraint genericity. We call it bounded genericity. Because of the differentiation between extends and implements, the bounded generic type declaration also makes this distinction:

```java
public class Stack<T extends Number>
public class Stack<T implements Comparable>
```

// multiple bounds

```java
public class Stack<T extends Object & Comparable>
```

It is important to know that genericity in Java is an extension of the Java programming language and not of the Java type system, i.e. the Java Virtual Machine (JVM). The generic type information will be lost during compilation [BOSW98]. Adding genericity to a programming language which is actually type-safe is dangerous, even worse if we do not extend the type system.

Assume we have the following code; we desire the situation in the first three lines [HBM+03]:

```java
List<String> sList = new List<String>();
List<Object> oList = new List<Object>();
oList = sList;
oList.add(new Integer(12));
```

Therefore, we should treat a list of strings as a list of objects. But we can also see the resulting catcall (Change Availability or Type):

```java
oList.add(new Integer(12));
```

This is not type-safe at all, thus we cannot accept this for the type system of Java. Eiffel does not care about this situation yet [HBM+03]. The Java compiler returns an error at the assignment `oList = sList;` to avoid a possible catcall. However, it would be desirable to have a generic super type. This makes sense if we want to prune all objects from a list or get a particular object in the list. Java introduces the wildcard type concept [BGT+04] to fulfill this wish:

```java
List<?> alist;
```

Then, the assignment `aList = sList;` is perfectly okay, but the call

```java
aList.add(new Integer(12));
```

is still impossible, because the compiler is still unable to determine the actual parameter of `oList`. It is not possible to determine the correct signature for `add(E obj)`. 
On the other hand, we can prune now all objects from the list or get an object at a particular position.

Java introduces wildcard types to overcome the problems of calls in its type-safe environment and still to allow access to the elements ignoring their actual type.

We have also the possibility to constrain wildcard types. It is possible to bound the generic parameter to be a supertype:

\[
\text{List} <\text{? super Square}> \ aList;
\]

**Reflection on genericity**

Java 1.5 uses a few new interfaces shown in figure 2.2 to provide reflection on genericity.

![Figure 2.2: New reflection on genericity interfaces in Java 1.5](image)

The meaning of those new interfaces are described in the following list:

- **GenericArrayType.** Array of a generic type, such as \( T[] \).
- **GenericDeclaration.** Entities that declare type variables, such as \( \text{List}<T> \).
- **ParameterizedType.** Parameterized type with an actual generic, such as \( \text{List}<\text{String}> \). We are only able to use them with *bounded genericity*.
- **TypeVariable \(<D>\text{extends GenericDeclaration}>.** Type variable, such as \( T \) in \( \text{List}<T> \).
- **WildcardType.** Wildcard type, such as \( \text{?} \) or \( \text{? extends Thread} \).

We can only get genericity information available in the class description, because the type system is not extended. Thus, it is not possible to get information about the actual type variable, e.g. \( \text{Integer} \) in \( \text{List}<\text{Integer}> \) will not be reflective visible during runtime.

The next example describes the reflection information on genericity available in Java 1.5:

```
public class Stack<T implements Comparable<? extends Number>>
```

The reflection API provides the following methods to get information about the class declaration above:
2.2 The Java reflection API

• Getting the type variable:

```java
Class claz = Stack.class;
TypeVariable[] tv = claz.getTypeParameters();
String name = tv[0].getName();
Type[] types = tv[0].getBounds();
```

name: \( T \)
upper bound: `Comparable<? extends Number>`

• Getting details about this upper bound, which is an instance of `ParameterizedType`:

```java
ParameterizedType pt = (ParameterizedType) types[0];
Type t = pt.getRawType();
```

Raw type: `Comparable`

• Getting details about the type variable of the declared `Comparable`. The type variable is an instance of `WildcardType`:

```java
types = pt.getActualTypeArguments();
WildcardType wt = (WildcardType) types[0];
types = wt.getUpperBounds();
```

Upper bound: `Number`

For more examples of genericity in Java see [San04].

2.2.5 Evaluation of the Java reflection API

Constructors vs. methods

We should first remember the class diagram in figure 2.1. Although a constructor is in a certain way a method, there exists no relation between the classes `Constructor` and `Method`. The reason is that the semantics of invoking a method or a constructor are different:

• Invoking a method (in class `Method`):

```java
public Object invoke (Object obj, Object[] args)
```

needs the knowledge of a particular object, `obj`, which has its own state.

• Invoking a constructor (in class `Constructor`):

```java
public Object newInstance (Object[] initargs)
```

does not need an object, because it is a class (static) and not an instance method.

A type representing both `Constructor` and `Method` can not have a single invocation method. Therefore, it would not make sense to take them into one class. However, if we want to invoke a static method of a class, we have to call:

```java
aMethod.invoke (null, aTypeArray);
```
The problem is that Java distinguishes between (instance) methods, static (class) methods, and constructors. But we could also treat constructors like static methods. Therefore, it would make sense to make a class describing the general behavior of a method.

We also suggest a new design: the separation of objects and classes in a reflection library. It makes more sense to access reflection information about instance members in `Object` and about class members in `Class` respectively, because the same differentiation is made when writing Java code.

`getDeclared...` vs. `get...

As we have seen in section 2.2.3, we are only able to get all declared members of the reflected class itself or only all public members including the one of its ancestors. Normally, a class cannot access private members of its parent classes. But, if we allow accessing private fields through reflection, there should be a possibility to obtain all members of a class.

On the other hand, getting all private members is possible. We can get all private members on each hierarchy level. We only have to loop over all parents up to `Object`.

We have to treat fields and methods separately when speaking of getting members. Java does not provide the methods `getMembers` and `getDeclaredMembers` respectively. This approach is not comprehensible, because there exists a general description of members.

**Special treatment of arrays**

Java uses the class `Array` which has only `static` methods to access information about an array. For instance, we call

```java
Object obj = Array.get(anArray, 11);
```

to get the object at a particular position in the array.

Every type in Java is declared as a `Class`. They wrap primitive types to their class types, like `Integer`, `Float`, etc. Therefore, they should consequently describe an array in a class. This approach would lead to the situation that `Array` inherits from `Class`. It is dubiously that Java needs a special treatment of arrays at all. It is a matter of how to implement an array type. If they described arrays in a normal class like in Eiffel, it would not be necessary to separate arrays for reflection access. They must provide is a facility to query a type whether it is an array type. If it was an array type, we could invoke its `indexing` for instance. Another example: every Java programmer knows to query the length of an array with `anArray.length`. The lack of a class describing an array makes it impossible to receive information about this field via reflection.

**Special treatment of primitive types**

The Java programming language does not treat primitive types, such as `int`, `double`, etc., as objects on runtime. Therefore, the class `Field` contains special getter and setter for the primitive types. This makes the use of the reflection API more difficult.

**Java 1.5**

The inability of getting generic instance type information during runtime prevents the consistency of the reflection API. In previous releases it was possible to determine the exact type of an object and instantiate a new one (not clone). Since the method `getActualTypeArgument` in the class `ParameterizedType` returns also wildcard types or type variables, its name is not chosen carefully - even wrongly.

The hierarchy of `Type` with its subinterfaces and the class `Class` (`Class` inherits from `Type`) is another point to discuss. First, a generic class declaration is rather a family of types or an infinite
set of types \[\text{Mey97}\] than a single type. We do not know anything about a type, until a generic derivation is available.

Second, why does \(\text{Class}\langle T\rangle\) inherit from \(\text{GenericDeclaration}\) and \(\text{ParameterizedType}\) does not, although a parameterized type is actually a generic declaration? And why does \(\text{Class}\langle T\rangle\) not just inherit from \(\text{ParameterizedType}\), since it should be a generic declaration? Parameterized types are reserved for generic bounds with actual parameters, but either way, we have to try if the actuals are wildcards, normal types or type variables. And if a parameterized type can have generic type variables, classes should also be parameterized types.

These reasons and the example in section 2.2.4 (p. 10) show that the new reflection API is not intuitive to use and obtaining information about genericity is cumbersome.

Figure 2.3 shows a better suggestion for the relation between types and classes.

![Figure 2.3: A better class/interface relation](image)

**Error handling**

The way of handling no result or not found is not consistent in the design of the reflection API - even in the whole Java SDK. Some methods return \text{null} to indicate no result, others throw an exception.

Take the class \(\text{Class}\) for example. If the search for a particular field, like

\[
\text{claz.getField("length");}
\]

fails, the method throws a \text{NoSuchFieldException}. In this case the return value \text{null} would make much more sense, particularly if real errors lead to exceptions too.

**Aspects on software quality**

The description of the reflection API of Java showed that we are allowed to get and set fields even if they are private. In some circumstances, there is now way around this matter, e.g. thinking of a debugger. The necessity of making private fields accessible with \text{field.setAccessible(true)}; reduces the risk of imprudent access. But then, the state of an object can still break its invariants and there is no guaranty that our piece of software works correctly.

The only possibility to prevent accessing private members is to start the Java Virtual Machine (JVM) with the command line option \text{-Djava.security.manager}. But this does not protect our code against foreign access.
Chapter 2. Existing Reflection APIs

2.3 The .NET Framework reflection API

First of all, the .NET Framework reflection API provides much more features than the one of Java. We can obtain most of the reflection information in Java, but it is partly easier in the .NET Framework.

This section does not give a full introduction to the use of the .NET Framework reflection API. Rather it exposes the differences of the library designs.

As from now, the .NET Framework is only called .NET to simplify matters. We have chosen the .NET programming language C# for the example codes.

2.3.1 Class design

The three namespaces System, System.Reflection and System.Reflection.Emit in the .NET class library belong to the reflection API. The class System.Type takes over the part of the class Class from Java.

![Figure 2.4: Inheritance hierarchy of System.Type](image)

Figure 2.4: Inheritance hierarchy of System.Type

Figure 2.4 visualizes the inheritance hierarchy of System.Type. Compared to Java, it strikes that the topmost class is not Type, but Type inherits from MemberInfo and implements IReflect:

- **MemberInfo**, Information about a member, like member kind (field, method, etc.), name, declaring type etc.

- **IReflect**, Methods to obtain information about members and to invoke them.

The Common Language Runtime (CLR) of .NET provides a derived class RuntimeType created only once per object at runtime.

The class Type is not the only hook to reflection. It is possible to load an assembly to inspect its modules, classes and members. The difference between a module and an assembly is that a module must not be an executable, e.g. a DLL. The hierarchy is: assembly, module, namespace, class. We are able to reflect all classes in a module or assembly. This is not possible in Java, i.e. we cannot retrieve information about the content of a package or a whole program.

A more interesting part of the API is the design of the method and constructor relation (figure 2.5).

.NET does not separate constructor and method. The class MethodBase describes common features, like modifiers or parameters of a method and provides methods to invoke them dynamically.

ParameterInfo describes formal parameters. They are not just types as in Java. The CLR provides different kinds of parameters, such as in, out or optional. We also want to get this information via reflection. It is additionally possible to ask the position of the formal parameter.

Since .NET has more language constructs as Java, e.g. properties and events, we can inspect them too (EventInfo, PropertyInfo).
2.3 The .NET Framework reflection API

![Diagram showing the relation between Constructor, Method, MethodInfo, and ConstructorInfo in .NET](image)

Figure 2.5: Constructor, Method relation in .NET

The special namespace `System.Reflection.Emit`

A very exciting feature of the .NET reflection API is the ability to write assemblies at runtime. The namespace `System.Reflection.Emit` contains classes to build an assembly from scratch, i.e., starting at an assembly and ending in writing Microsoft Intermediate Language (MSIL) instructions directly. For an example use of `System.Reflection.Emit` refer to [Lib03].

### 2.3.2 Example use of the API

Section 2.2.3 describes in which situations reflection can be useful. The example in this section is only demonstrating how we use the specialties of the .NET reflection API.

It is possible to invoke members directly within the class `Type`. (Actually there are three possibilities, but one is abstract and the other has an additional parameter of type `CultureInfo` describing a human culture including its calendar system etc.):

```csharp
public object InvokeMember (string name,
    BindingFlags invokeAttr,
    Binder binder,
    object target,
    object[] args)
```

The binder is a type converter. It is mostly left `null` indicating to use the default binder. The real interesting part of this method is its parameter `invokeAttr` of the enumeration type `BindingFlags`. We can declare which kind of member we want to search, how a member should be searched and how we would like to invoke the member.

Taking the example from the section 2.2.3 we invoke the member `setAge (int age)` directly with:

```csharp
InvokeMember ("setAge", BindingFlags.InvokeMethod | BindingFlags.Public,
    null, person, new object[] {"John"});
```

We can also set the name directly:

```csharp
InvokeMember ("name", BindingFlags.SetField | BindingFlags.NonPublic,
    null, person, new object[] {"John"});
```
The class `Type` provides a similar mechanism to search for members:

```csharp
public MemberInfo[] FindMembers(MemberType memberType,
      BindingFlags bindingAttr,
      MemberFilter filter,
      object filterCriteria)
```

Unlike in Java, it is for instance possible to search a member with a pattern:

```csharp
FindMembers(MemberType.Field, BindingFlags.NonPublic, Type.FilterName, "Set*")
```

With this mechanism we can find more members than with Java, e.g. `BindingFlags.NonPublic` means all non-public members - even the ones from inherited classes. `getDeclaredFields` in Java corresponds to `BindingFlags.NonPublic | BindingFlags.DeclaredOnly` in .NET.

The .NET reflection library is able to do everything that the Java reflection API can do, but not vice versa.

### 2.3.3 Forthcoming .NET Framework release

**Genericity**

The use of genericity in .NET looks similar to the one of Java. But there are some differences: .NET does not provide wildcard types and constraint genericity is a bit different declared. An advantage of .NET is to allow generic derivations with built-in (primitive) types.

Since .NET does not support wildcard types, they do not allow covariance in generic parameter types to fulfill type safety. Thus, in .NET is no notion of a generic supertype. Because Eiffel for .NET supports covariance, the compiler annotates generic parameters in the Common Intermediate Language (CIL). But so far, Eiffel for .NET is the first and only programming language using this feature.

For more information about genericity in .NET 2.0 see [KSI01] and [Low03].

**Reflection on genericity**

The introduction of genericity in .NET 2.0 is an extension of the CLR and not only of the language. This major advantage makes it possible to determine the actual generic parameters of a generic derivation.

Microsoft is designing the reflection mechanism on genericity completely different than Sun. First, they are not introducing new classes, they are only extending the class `Type` with additional members. This results in a simple use. Second, Microsoft names the difference between an actual and a formal generic parameter a generic and an unbounded generic parameter.

**Example**

Assume we have the following generic class and a particular derivation:

```csharp
public class List<T> where T : IComparable

public List<string> list = new List<string>();
```

We can reflect on these two instructions as follows:

- `Type type = list.GetType();`
- `type.HasGenericParameters; → true`
2.3 The .NET Framework reflection API

- `type.GetGenericParameters();` → Array of `Type` containing `System.String`
- `type.GetGenericTypeDefinition();` → The `Type` of `List<T>`

Going further with `type = type.GetGenericTypeDefinition();`

- `type.HasGenericParameters;` → true
- `type.HasUnboundGenericParameters;` → true
- `type.IsGenericTypeDefinition;` → true
- `type.GetGenericParameters();` → Array of `Type` containing `T`

We may derivate a generic type at runtime:

```csharp
Type type = Type.GetType("List");
Type[] types = {typeof(string)};
Type boundedType = type.BindGenericParameters(types);
```

The object `boundedType` describes the same type as the first `type` above.

2.3.4 Evaluation of the .NET reflection API

The .NET reflection API provides more reflection facilities than the one of Java, but its design is not much better.

**MemberInfo vs. Type**

The relation between the classes `MemberInfo` and `Type` is a bit dubious. It makes no sense to let a type be a member. Their relation in .NET is contradictory to their semantics. A type and a member are two separate constructs. We would not have any loss of information if we separated them completely.

In another part, .NET makes exactly the reverse. A member has always an access modifier, like `public` or `private`. But the class `MemberInfo` does not provide any modifier information at all. This information is only available in `FieldInfo` and `MethodBase`, but not in `EventInfo` and `PropertyInfo`. This may be the reason for the missing query possibility in `MemberInfo`. This is rather incomprehensive, because we can even declare a property or an event as private or public, but particularly because the default access modifier is `private`. The reason that a property is almost always declared public, may be acceptable for this design decision. However this does not map the specification of the .NET Framework to its reflection API.

.NET does not make a distinction between the semantics of class and instance members in its reflection facilities either (see section [2.2.5](#) on p. [12](#)).

**No special treatment of arrays**

In section [2.2.5](#) we criticize that Java does not let the class `Array` be a subclass of `Class` and therefore cannot be reflected like other types. .NET solves this situation better. The class library contains a class `Array` declaring the behavior of an array, but it does not derive from `Type`. Every array object, like `string[] str;` is an instance of `Array`. The same applies for delegates and enums. It is a good approach to define classes describing each CLR construct. Eiffel is using this approach too.
No special treatment of primitive types

In comparison to Java, we do not need to treat primitives types specially when getting or setting them via reflection. The CLR wraps them to the corresponding object types automatically.

.NET Framework 2.0

The approach to resign the introduction of new classes to handle reflection on genericity is okay, but not fully satisfying. As we have discussed in section 2.2.5 a generic class is rather an infinite set of types than a single type. This design approach of .NET makes the use of the API easy, even though it is not desirable.

If we compare the reflection API of .NET with the one of Java, it is at first glance obvious that it has more features. But with some workaround, many of the reflection information in .NET are also available in Java.

Error handling

The error handling in .NET is intuitive. If a searched method or field cannot be found, a null reference is returned. This is much better solved as in Java (see 2.2.5 on p. 13).

Aspects on software quality

Protection of our own code against reflection access is not possible in .NET either. A system administrator can give certain access permissions on assemblies. Additionally, the user must have the permissions to access reflection information about non-public members.

2.4 Reflection in Smalltalk-80

Almost everything in Smalltalk is somewhat different compared to Java and .NET, as we will see later. But first, we need a short introduction to Smalltalk to understand its reflection mechanism and special characteristics.

2.4.1 A short introduction to Smalltalk

In Smalltalk most of the object-oriented constructs are named different than in Eiffel or Java. We call a method call a message. The receiver of the message is the target of the method. We call the desired operation selector which names the method and is also the list of the argument names. This is a bit confusing as it enforces us to name a method only with its signature, but without type information. Here is an example of a method:

\[
\begin{align*}
setX &: xValue \\
setY &: yValue \\
x &\leftarrow xValue. \\
y &\leftarrow yValue
\end{align*}
\]

The selector is \(setX:setY\): which reflects the name of the method. We call the whole string \(setX: xValue setY: yValue\), thus the method declaration, the message pattern. In Smalltalk the assignment operator is \(\leftarrow\), such as \(\_\_\) in Eiffel. In Squeak [KBI], which is an open source Smalltalk system, we type an underscore ‘\_' to print a \(\leftarrow\).

The receiver of a message is always an object. In Smalltalk only qualified calls are accepted. We must take \(self\) if the message should be sent to itself (\(this\) in Java or \(\_\_\); \(Current\) in Eiffel). We do not use a . (dot) for qualified calls:
Smalltalk distinguishes static and non-static variables between class and instance variables. The same applies for methods.

In Smalltalk we use block arguments to implement functions to be applied to all elements of a data structure. Suppose we have a set of integers \( set \) and want to add up all elements in \( set \):

\[
\begin{align*}
\text{sum} & \leftarrow 0. \\
\text{set} \ do: [ :\text{elem} | \text{sum} \leftarrow \text{sum} + \text{elem} ].
\end{align*}
\]

We use symbols to represent a string. For instance: \#bill is the representation of 'bill' (strings in Smalltalk are single quoted).

### 2.4.2 Reflection facilities

The most important issue to know about Smalltalk is its absence of static type checking. Smalltalk checks everything, except the syntax, during runtime.

Smalltalk is almost entirely written in itself. Therefore, nearly each language construct is an object. Primitive types, for instance, like \texttt{Integer} or \texttt{UndefinedObject} (void or null), do also have a class. Even their basic operations, like +, have their own methods.

The compiler generates an instance of a \texttt{CompiledMethod} for every compiled method and the system generates a \texttt{MethodDictionary} object for every new class. The system needs it for method lookups during runtime. Unlike Java or .NET, Smalltalk needs its own reflection facilities to work.

### Class design

Next, we discuss the concept of meta classes and classes to get a deeper look how Smalltalk manages object creation. The fact that Smalltalk is written in itself yields to a problem. Which object should be used if we access class variables or methods? The answer is with meta classes. For every class exists a \texttt{Metaclass} whose instance is or describes the class itself. Therefore, every class is an instance of its meta class. On the other hand, in Smalltalk exists a class \texttt{Class} describing an object. Thus, every object is an instance of a class.

\texttt{Metaclass} and \texttt{Class} inherit from \texttt{ClassDescription}, since they are descriptions of class-like constructs. \texttt{ClassDescription} inherits from \texttt{Behavior}. Figure 2.6 shows this class hierarchy.

![Class/meta class kernel](image)

**Figure 2.6: Class/meta class kernel**

The scope of the duties of these four classes according to [CR89] is described in the following list:
- **Behavior** defines the minimum state necessary for objects that have instances, including: class hierarchy link, method dictionary, and description of instances in terms of the number and the representation of their variables.

- **ClassDescription** represents class naming, class commenting and naming of instance variables. It adds structures to organize the selector/methods pairs of the method dictionary. They are referred to as message protocols or categories. This is called feature categorization in the Eiffel notation. The same can be done with classes named class categorization.

- **Class** instances describe the representation and behavior of objects. It adds more programming support facilities to the facilities provided in Behavior and more descriptive facilities to the ones provided in ClassDescription.

- **Metaclass** provides protocols for initializing class variables and instance creation of the described class.

**Specialties of the reflection facilities**

Smalltalk provides a large number of features to reflect objects and classes. Each feature in the reflection APIs of Java and .NET is covered. Now, we look at several special features of the described classes.

We should keep two general things in mind. First, since we name almost everything different in Smalltalk, the message pattern of the reflection facilities are different too. Second, assume there exists a class **Person** and a corresponding instance **person**. To get the class of an object in this case, we simply use **person class** or **Person**. We receive the meta class with **person class class** and **Person class** respectively.

**Behavior**

- **allCallsOn: selector, allLocalCallsOn: selector**
  Returns all methods calling selector in the whole hierarchy or only those declared in the receiver.

- **allSubclassesDo: aBlock**
  Evaluates aBlock with each subclass of the receiver. Example: assume we want to count all subclasses of **Collection** that have the selector **addFirst**.
  
  ```plaintext
  i ← 0.
  Collection allSubclassesDo: [:
      |class | (class includesSelector: #addFirst:) ifTrue: [i ← i + 1]]
  ```

- **sourceCodeAt: selector**
  Returns the source code of selector in a string.

**ClassDescription**

- **allMethodsInCategory: aName**
  Returns a list of the methods in category aName. This includes the methods of the receiver and of receiver’s ancestors.

- **copyCategory: cat from: class**
  Copies each message found in category cat from the class class to the protocol of the receiver.

- **renameInstVar: oldName to: newName**
  Renames the instance variable referred by oldName into newName.
2.4 Reflection in Smalltalk-80

**Class**
- `addInstVarName: aString, removeInstVarName: aString`
  Adds, removes an instance variable to, from the receiver. The same can be done with class variables.

2.4.3 Evaluation of the reflection facilities of Smalltalk

Smalltalk, except for its absence of static type checking, is the language coming closest to Eiffel compared with the languages discussed so far. There are categories, no special meaning of constructors (every method can be a constructor), every primitive type has its own class with the corresponding operations, no more than one method with the same name (message pattern), and so on. What lacks completely is the notion of a type and therefore the differentiation between types and classes. This is not a problem because Smalltalk does not know genericity. Finding and invoking methods is much simpler, because we do not need to know anything about their signatures. After all the class design shown in figure 2.4.2 is simple and obvious.

Since Smalltalk is written in itself and it describes every behavior in a class, it does not solve issue the meta class of the meta class satisfiable. We describe this situation a bit deeper.

**Object and class hierarchy**

Smalltalk generates a meta class for every class automatically. To overcome the problems of meta class compatibility described in [Gra89], meta classes must have the same hierarchy as their corresponding classes. To remember, there are three levels describing an object: the object itself is an instance of its class which is in turn an instance of its meta class. This description of [Gra89] is, in the sense of exact semantics in the object-oriented paradigm, not satisfiable at all. The phrase an object is an instance of its class means in Smalltalk that an object is an instance of the corresponding instance of Class.

A curious situation can be found in the three levels of instances. The meta class of a meta class is again a meta class. That comes from the approach to represent meta data in an object and not, like in .NET’s CIL, in form of written data that we can query by parsing the text. After all, this is not a problem at all, we only have to know that querying a class of a class in a loop never terminates.

The three levels are described in [GRS9], also known as the Smalltalk-80 schema.

Smalltalk is the only language so far which makes the distinction between class and instance members at the level of reflection.

Only one negative approach has attracted our attention while looking at the reflection mechanism of Smalltalk. The class `Object` provides a selector `instVarAt:put:` to change the value of an instance variable. A corresponding selector for class variables is not available.
2.5 Reflection in scripting languages

Normally, scripting languages are interpreted. It is obvious that reflection takes over a fundamental part of the interpreter, because they need lookup tables or dictionary to find methods or fields. Therefore, it would not be so hard to add reflection facilities to a scripting language.

2.5.1 Ruby as an example

Ruby, a newer object-oriented scripting language, is a good example. The reflection API of Ruby is similar to the ones we have discussed in the previous sections. It seems at first glance that Ruby makes the distinction between getting information about an object or a class. We obtain information about an object from the interface of Object and information about the class itself from the interface of Class.

In Ruby we are not allowed to set variables directly - not even with reflection. Additionally, Ruby does not provide a mechanism to add a new method to an object during runtime, but we can merge a given module into an object. Thus, writing variables is possible, but exhausting and not completely dynamically. We can get and set constants directly.

There is something irreproducible in the reflection API of Ruby. The class can be queried to get the names of all instance methods, but not of instance variables. However, we can do this in Object. Since it is also possible to collect the methods in Object, it is not necessary to provide queries about instance methods in Class. This particular situation leaves the question open, if the designer of Ruby’s reflection API wanted to make the distinction between classes and objects.

2.6 Existing reflection in Eiffel

Eiffel does not miss reflection completely. Two possibilities of retrieving runtime information are provided. First, the need of describing agent expressions and second, the Kernel Library class INTERNAL.

2.6.1 Describing agent expressions

Agent expressions need corresponding objects to separate construction time from call time. The Kernel Library classes ROUTINE, PROCEDURE and FUNCTION are used to make that separation:

- ROUTINE (deferred) provides features to query the operands and the target, to access pre- and postconditions and to call the routine.
- PROCEDURE derives from ROUTINE and provides an effective call feature.
- FUNCTION derives also from ROUTINE, but has an additional formal generic for the result type and provides an effective call feature.

The queries to check pre- and postconditions are not implemented because the Eiffel compiler and runtime system do not support these information yet.

2.6.2 Kernel Library class INTERNAL

INTERNAL offers several features to access and manipulate the state of an object and to create new instances of a particular type. We will not go through the features now; they will be discussed in-depth in the next chapter.
2.7 Conclusion

As we have seen so far the support of reflection differs strongly in existing programming languages. Some common features, such as obtaining all members, are represented throughout (with some restriction in Eiffel). Smalltalk supports the largest set of reflection facilities, primarily because the interpreter of the language needs reflection to work. This is amazing, as Smalltalk is over 20 years old.

<table>
<thead>
<tr>
<th>Facility</th>
<th>Java</th>
<th>.NET</th>
<th>Smalltalk</th>
<th>Ruby</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine the type of an object</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Information about a deferred type</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Search fields by name</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>∼ ⋆</td>
</tr>
<tr>
<td>Get and set field values directly</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>∼</td>
<td>⋆</td>
</tr>
<tr>
<td>Search methods by name</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Invoke methods</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Information about formal arguments</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>∼ ⋆</td>
</tr>
<tr>
<td>Get superclass</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Information about package/assembly</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Dynamic object instantiation</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Generate methods dynamically</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Add methods dynamically</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Separation of class and object</td>
<td>×</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Get formal generics</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Get actual generics</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Get generic constraints</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Derivate type</td>
<td>×</td>
<td>⋆</td>
<td>⋆</td>
<td>×</td>
<td>⋆</td>
</tr>
</tbody>
</table>

* available; o available with constraints; ∼ possible with workaround; × not available

*Either public incl. superclasses or private excl. superclasses

*Fields are described by their number in the class

*Only in conjunction with agents

*Without name

*Even subclasses

/Only via new

*Without initialization

*Even classes or assemblies

*Only via merging

/Not consistent

*Attention, actual generic count is unavailable
2.7.1 Expected set of reflection facilities

Table 2.1 gives a comparison of the reflection facilities of each discussed programming language and runtime environment.

We take Smalltalk as the complete set of reflection facilities. This also includes, together with strict separation of class members and instance members, generating classes and its members dynamically. Thus, it should be possible to use every static language construct with reflection during runtime. Static is not enough, because we miss important runtime information. Thinking of dynamic binding, the exact runtime type of an object and its state could not be known statically. We can tell nothing about an object, without this information. If we know that, all other information which we can see inspecting the source code is available: constructors, methods, fields, superclass, subclasses, which classes have a field of this type, and so on.

Until now, runtime features are not considered either: dynamic method invocation, dynamic object instantiation or loading, dynamic type derivation, etc. So, maybe the demand every static language construct is too much. It is not necessary to be able to retrieve the whole source code at runtime. Member declaration and their invocation is enough.

We do not discuss whether reflection is a part of the programming language or a part of the runtime system.

2.7.2 Expected design of a reflection API

The class design of a reflection API should reflect the concepts of the object-oriented paradigm. For example, the figure 2.2 shows how the design works against our o-o understanding from [Mey97] (beside: this example also leads to a complex use).

Once more, the separation of class and object makes a lot of things easier. In general we do not exactly identify the type of an object until runtime, because of dynamic binding and particularly in the case when genericity is available. Nevertheless, we can invoke class members without runtime information.

We have two ways to implement this separation approach. Either we make all reflection information about objects available in the topmost class, Object in Java or .NET, ANY in Eiffel and those about classes in Class (cp. Java in 2.2.2); or we design a class architecture covering the object model of the corresponding programming language. Then, the topmost class only gives a hook into the reflection library. Considering reflection in Eiffel, it is better to use the second approach, because we do not want to touch the class ANY.

After all, we expect that a reflection library should reflect the object model of a programming language and its runtime environment as exact as possible, but with respect to usability.
Chapter 3

Reflection for Eiffel

3.1 Introduction

In the previous chapter we gained an insight how existing reflection APIs are designed. Only the reflection library of Smalltalk is satisfactory. We are rather bothered by the designs of the libraries than with their functional range. Their use is sometimes cumbersome. But primarily, their designs reflect rarely the semantics of the corresponding programming language and the object-oriented concepts respectively. A typical example is the strange model of the different types in Java 1.5 (figure 2.2).

We need a reflection library for Eiffel because INTERNAL does not support all expected reflection facilities. We define the expected set of reflection facilities later.

Since Eiffel has a very sophisticated object model, we must design a reflection library that maps this model as exhaustively as possible. The designs of the evaluated reflection APIs do not follow this approach. On the other hand, an exhaustive model would lead to taxomania. We will see examples later where taxomania would occur, if we claimed a correct model in respect of the o-o concepts introduced in Eiffel. We have found a way to satisfy both requirements.

In this chapter we define the term reflection and what we expect from the design of a corresponding library. We also specify a minimal set of reflection facilities that a reflection library must provide. After that, we show how we have designed the Eiffel Reflection Library (ERL) and implemented parts of its features. Since Eiffel does not provide full reflection yet we were restricted to the scope of functionality of INTERNAL.

3.2 Requirements

The object model of Eiffel is very sophisticated. It provides various concepts to ease programming and to improve the quality of the software. The Eiffel Reflection Library should reflect these concepts so that we are able to retrieve structural relations between objects, their types and their classes at runtime. For instance, we want to be able to discover via reflection how the programmer of a system has adapted features along a class hierarchy.

It is not clear if code emitting belongs to reflection. We think reflection should only act on and show existing data. Therefore, we do not consider code emitting in ERL. We define the term

\footnote{Taxonomy mania, Mey97}
Reflection is the ability to retrieve exact type information about objects at runtime of a system, to query the interface of a type, to invoke its members dynamically and to derivate new types from (generic) classes.

A library providing reflection must reflect the object-oriented concepts of the corresponding programming language as exhaustively as possible in respect of usability and taxonomy.

Defining what reflection is and how we have to design it leaves the question open, which functionality it should provide to fulfill the description above. Accordingly, we must specify a minimal set of reflection facilities provided by a reflection library.

3.2.1 Minimal set of reflection facilities

We must distinguish between getting reflection data from the runtime system and extracting relevant information. We have to think about this issue by asking questions like: Is the reflection library responsible to return all attributes of a type or is it the duty of the user to extract them out of the set of all features?

With such questions we are able to define a minimal set of facilities that a reflection library must provide to get as enough information as we have described in table 2.1.

We have chosen a minimal set of reflection facilities to satisfy this request:

<table>
<thead>
<tr>
<th>Language element</th>
<th>Queries</th>
<th>Status</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universe</td>
<td>all clusters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all types</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>class for name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type for name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type of an object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>actual generic parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>base class</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>feature clauses</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>invariants</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lookup binding</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>has actual generics</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new instance</td>
<td>is basic type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parents</td>
<td>is deferred</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is expanded</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is obsolete</td>
<td></td>
</tr>
</tbody>
</table>
3.2 Requirements

We give a detailed description of each of these features in the next sections.

Compared to the reflection APIs discussed in the previous chapter, some well known reflection functionalities are missing. It should be possible, for instance, to search for attributes by name. But such a functionality does not belong to a minimal set, because in this particular example, it is possible to get the required attributes by looping over the set of features.

The minimal set specifies enough features to do even complex queries such as searching all types that have an attribute of a particular type or getting all instances of a given type.

Figure 3.1 visualizes the class design which we have chosen to satisfy the demand of a meaningful library according to the definition above. We will discuss the various kinds of features and their integration in ERL later.

### Table 3.1: (continued)

<table>
<thead>
<tr>
<th>Language element</th>
<th>Queries</th>
<th>Status</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster</td>
<td>classes</td>
<td>is root cluster</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>cluster</td>
<td>is generic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>formal generic parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type with actuals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic parameter</td>
<td>constraints</td>
<td>has constraints</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td>bound</td>
<td>has creator constraints</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature clause</td>
<td>category</td>
<td>has clients</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clients</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>features</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>feature clause</td>
<td>is deferred</td>
<td>set object</td>
</tr>
<tr>
<td></td>
<td>effective parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>names</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>parents</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>preconditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>postconditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>position</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routine</td>
<td>formal arguments</td>
<td>is creator</td>
<td>call</td>
</tr>
<tr>
<td></td>
<td>operands</td>
<td></td>
<td>set operands</td>
</tr>
<tr>
<td>Procedure</td>
<td></td>
<td>is creator</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>item</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>attribute type</td>
<td></td>
<td>replace</td>
</tr>
<tr>
<td></td>
<td>item</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
At the end, ERL will be a reference implementation of the minimal set of reflection facilities and the suggested class diagram.

Figure 3.1: Main parts of ERL
3.3 Design of ERL

Designing a reflection library means not only to choose a set of functionality carefully, but also to design a sound model (figure 3.1). We will see why a reflection library mapping the Eiffel object model would make some parts of the class hierarchy too complex.

Every class of the library has the prefix ERL because we are not allowed to use keywords and existing Eiffel classes. Since Eiffel is not case sensitive, it is impossible to name a class FEATURE or CLASS. Examples of existing classes are ROUTINE or FUNCTION used for the agent mechanism (see section 2.6.1). It is also convenient to see which classes form a library.

ERL considers the object model of ETL3. ECMA is working on a new Eiffel standard (ETL3) of which a draft was available for this thesis.

3.3.1 Separation of type and class

The separation of classes (types) and objects is insufficient when dealing with genericity. A class with generic parameters describes a set or a family of types. A type is a derivation of a class with actual generic parameters and without a particular state. An object is an instance of a type with a state satisfying its invariants.

Now the question arises which language constructs belong to a type and which to a class. We use each construct when writing classes, but it makes more sense to let a few of them belong only to the type.

Features

Since routines can have generic arguments, they are not completely defined until declaration time:

```eiffel
class LIST [G]
feature
  put (v: G) is
    do
    end
end -- class LIST
```

We are unable to determine the type of that routine until we declare a derived type of LIST, e.g. list: LIST [STRING]. Thus, it makes not much sense to let routines be a part of a class. Not all routines have generic arguments, but there is no practical reason to separate routines into the categories with generic arguments and without generic arguments.

We argue the same way for attribute types.

Creators

Creators have not a special treatment as constructors in Java. They are actually initialization procedures. If a creator is exported to ANY, we can also call it after initialization. Thereby and since they are routines, they also belong to the types.

\[2\text{Eiffel 4.80-00-10, 3 June 2002}\]
Chapter 3. Reflection for Eiffel

Feature clauses

We use a base class to define the export status of a feature clause:

```eiffel
feature {LIST} -- Element Change
```

If we argue the same way as with routines, the feature clauses should belong to the class. To keep things simple, we let them belong to the type because they are related to their feature.

3.3.2 Type

An object is an instance of a particular type which is determinable in every state of the system execution. The class `ERL_TYPE` represents the reflection information about a type.

Name

A type is completely identified by its name.

```eiffel
name: STRING
```

The name also includes the actual generics of the type, e.g. `STRING` in `LIST [STRING]`. Because generic derivations differ in their actual generics, they must belong to the type name, otherwise it is not clear of which type we are talking about. We speak about the base class of a type if we omit the actual generics.

Since a type is declared by its name, information about a type should be available without an object, particularly in case of deferred type.

Base class

The concepts `class` and `type` are separated if a programming language provides genericity. In this case, a type has a base class from which it is generically derived. A non-generic type is its own base class. If we want to be strict, we must make this separation in ERL, i.e. a type has only a base class if it is generic.

In Eiffel exists the notion of clusters. We may compare them with packages in Java or namespaces in C#. Eiffel has introduced clusters to group classes. We use them to ease programming. Because clusters are not part of a running system, we could omit them from reflection. However, sometimes we want to be able to find out in which cluster a class resists and what other classes the cluster contains.

We only need the class of a type if it is generic. Anyway, we need it for the cluster. Now we have to decide how we should model generic and non-generic classes. An obvious solution would be to have two kinds of classes. One with and the other without generic parameters. But which heirs from which? A non-generic class is a special kind of class. Nevertheless, it would be much simpler to let the non-generic class be the topmost in the hierarchy and to add additional features to the generic one. We have to consider another point. If we let the generic class be the topmost, we would have the danger of catcalls (Change Availability or Type): we had to hide (export to `NONE`) the query `formalGenerics` in the heir describing non-generic classes. The other way round is not better, because we would need to do an assignment attempt in order to call the query.

The simplest way is to have only one class to describe the class concept. This class has a flag to state if it has formal generics or not. We have decided that a type has always a base class.

We can use the same argumentation to explain why we do not distinguish between types with and types without actual generic parameters.

We can get the base class of a type with the query:

```eiffel
base_class: ERL_CLASS
```
3.3 Design of ERL

Lookup binding

One of the specialties of Eiffel is its sophisticated multiple inheritance support; primarily the ability of redeclaring, undefining and selecting features in an inheritance step. With

\[
\text{lookup\_binding (a\_feature: ERL\_FEATURE): ERL\_FEATURE}
\]

we get the feature in the reflected type, which would be invoked, if we called \text{a\_feature} in a parent type. Assume we have the hierarchy shown in figure 3.2.

Assume further we have the following code:

\[
\begin{align*}
u &: \text{ERL\_UNIVERSE} \\
t &: \text{ERL\_TYPE} \\
f, bf &: \text{ERL\_FEATURE} \\
\text{create } u \\
\text{u.type\_for\_name ("A")} \\
f &: \text{t.find\_feature ("f")} \\
t &: \text{u.type\_for\_name ("B")} \\
bf &: \text{t.lookup\_binding (f)}
\end{align*}
\]

In this case the query \text{lookup\_binding} returns the feature to call in \text{B} if we invoked \text{f} on a reference of type \text{A} attached to an object of type \text{B}. That way, we simulate the instructions

\[
\begin{align*}
a &: \text{A} \\
\text{create } \{B\} a \\
a.f
\end{align*}
\]

without creating an object and executing the code in \text{f}.

Feature clauses

We group the features of a type in feature clauses. The class \text{ERL\_FEATURE\_CLASSES} represents feature clauses and provides the queries:

- \text{category: STRING}. Feature clause category.
- \text{clients: SUBSET [ERL\_CLASS]}. Clients to which its features are exported.
- \text{features: SUBSET [ERL\_FEATURE]}. All features of this feature clause.
- \text{type: ERL\_TYPE}. Type to which this feature clause belongs.
- \text{has\_clients: BOOLEAN}. Is this feature clause exported to a special set of clients?
Chapter 3. Reflection for Eiffel

Features

Every feature of a type belongs to a feature clause. As defined in the minimal set of reflection facilities in table 3.1 they are only available implicitly in a type. Nevertheless, \texttt{ERL\_TYPE} provides the query

\texttt{features: SUBSET [ERL\_FEATURE]}

to make it less painful for us to acquire the needed information.

Java (section 2.2.3) distinguishes between declared members introduced only in the reflected type, and public members along the type hierarchy. In .NET we can get either all public members or a special selection given by a \texttt{BindingFlag} (section 2.3.2).

Beside features exported to \texttt{ANY} (public) and \texttt{NONE} (private) Eiffel allows a third export possibility: exported to a special set of clients: \texttt{feature \{} \texttt{STRING} \}, for instance.

Since reflection breaks the boundary between an interface and its implementation, we may argue that with reflection we should be able to get all features of a type including also the private features of the parents.

Finally, we decided that we want to be able to indicate from which client context we look at the type. According to the options operands principle described in \cite{Mey97}, we can set the context to a certain class and then demand all features exported to this class. The default value is the class \texttt{NONE} to declare that all features are demanded. The corresponding routines are:

- \texttt{client\_context: ERL\_CLASS}
- \texttt{set\_client\_context (a\_client\_context: ERL\_CLASS)}

Now, we have satisfied the minimal set of feature queries, but we additionally added some more queries to make it easier for a developer to retrieve attributes, routines, and deferred features of a type.

- \texttt{attributes: SUBSET [ERL\_ATTRIBUTE]}. All attributes of the reflected type.
- \texttt{routines: SUBSET [ERL\_ROUTINE]}. All effective routines of the reflected type.
- \texttt{deferred\_features: SUBSET [ERL\_FEATURE]}. All deferred features of the reflected type.

These are the primary groups of features: deferred and effective which are split into routines and attributes.

At last, we provide a function to search a feature by name:

\texttt{feature\_by\_name (a\_name: STRING): ERL\_FEATURE}

New instance

A required feature is creating new instances of a reflected type during runtime. There are two possibilities to instantiate a type:

1. We choose a creator and ask for a new instance initialized by the chosen creator.
2. We ask for a new instance without initialization and therefore, without a state fulfilling the invariants. We can afterward initialize the object by calling one of its creators.

The first attempt is intuitively more correct, because we do not let an object in a state without fulfilling its invariants. But since we are in a reflection environment, it is sufficient to let us choose afterward whether we want to initialize the object or not.

In ERL we ask for a new instance with:

\texttt{new\_instance: ANY}
Miscellaneous

- \textit{actual generics:} \texttt{LIST [ERL\_TYPE]}. A list of the actual generic parameters of this type, if it has actual generics.

- \textit{invariants (object: ANY):} \texttt{BOOLEAN}. True if \textit{object} fulfills the invariants of this type, false otherwise.

- \textit{parents:} \texttt{BAG [ERL\_TYPE]}. All direct parent types of this type.

- Type status reports:
  - \textit{has actual generics:} \texttt{BOOLEAN}. Has type actual generics?
  - \textit{is basic type:} \texttt{BOOLEAN}. Is type a basic type (such as \texttt{INTEGER, REAL} etc.)?
  - \textit{is deferred:} \texttt{BOOLEAN}. Is type deferred?
  - \textit{is expanded:} \texttt{BOOLEAN}. Is type expanded?
  - \textit{is obsolete:} \texttt{BOOLEAN}. Is type obsolete?

State of a type

A type has always a non empty name and a base class. If the type is \textit{ANY}, it has at least one parent type. Obviously, a type has actual generics if its base class is generic. A type has a set of features, if it has at least one feature clause.

3.3.3 Class

Every type has a base class; sometimes it is its own base class. The corresponding class in ERL is \texttt{ERL\_CLASS}.

Name

The formal generic parameters do not belong to the name of a class. A class name is unique. The corresponding query is:

\texttt{name: STRING}

Cluster

Clusters exist to keep things in a software system or library partitioned. We can query the reflected class to get its cluster:

\texttt{cluster: ERL\_CLUSTER}

In ERL, clusters are instances of \texttt{ERL\_CLUSTER} and provide the following queries:

- \texttt{name: STRING}. Absolute name of this cluster.

- \texttt{parent: ERL\_CLUSTER}. Parent cluster of this cluster.

- \texttt{classes: SUBSET [ERL\_CLASS]}. All classes in this cluster.

- \texttt{is root cluster:} \texttt{BOOLEAN}. True if this cluster is a root cluster.

A cluster has a parent cluster, only if it is not the root cluster.
Generic parameter

A generic class has a list of formal generic parameters:

\[ \text{formal generics: LIST [ERL\_GENERIC]} \]

A single generic parameter can additionally have a list of constraints. Moreover, each constraint can have restricted creators:

\[
\text{class } D [G \rightarrow \{ \text{CONST1, CONST2 create cp1, cp2 end} \}]
\]

The class \text{ERL\_GENERIC} describes a formal generic parameter. It provides the following queries:

- **name: STRING.** Name of the formal generic parameter.
- **constrains: LIST [ERL\_CONSTRAINT].** List of the generic constraints.
- **has\_constraints: BOOLEAN.** Has this parameter generic constraints?

A generic constraint can then be a type

\[
\text{class } D [G \rightarrow \text{LIST [STRING]}]
\]

or a class

\[
\text{class } D [G, L \rightarrow \text{LIST [G]}]
\]

The consequence of this issue is that we have to query if the generic constraint is a type or a class. In case of a type, the class of the generic constraint is its base class.

The class dealing with generic constraints is \text{ERL\_CONSTRAINT}. It provides:

- **class\_bound: ERL\_CLASS**
- **type\_bound: ERL\_TYPE.** Bound of the generic constraint.
- **creators: SUBSET [ERL\_ROUTINE].** List of the corresponding creators.
- **has\_creator\_constraints: BOOLEAN.** Has this constraint creator constraints?
- **is\_bound\_type: BOOLEAN.** Is bound a type?

The state of a constraint is clear now: it is always bounded to a class and if the bound is a type, then the class constraint is its base class.

Derive a type

Inspired by .NET 2.0, we also want ERL to instantiate types for us. In .NET the method to use is:

\[
\text{Type BindGenericParameters (Type[] typeArgs)}
\]

The corresponding query in ERL is:

\[
\text{type with actuals (a\_actuals: LIST [ERL\_TYPE]): ERL\_TYPE}
\]

The list of actuals must fulfill the given constraints of the reflected class:

\[
\text{conform generic constraints (a\_actuals: LIST [ERL\_TYPE]): BOOLEAN}
\]

If the class is not generic, we instantiate a type with the name of the class and the query

\[
\text{type for name (a\_name: STRING): ERL\_TYPE}
\]

in \text{ERL\_UNIVERSE} (see 3.3.8).
3.3 Design of ERL

3.3.4 Features in Eiffel

Two feature categories are declared in [Mey97]: first, the classification into command and query and second, into computation and memory. This classification is clarified in figure 3.3.

The figure considers only effective features, but Eiffel knows deferred (abstract) features too. Therefore, we should include deferred features in those categories. The consequential class diagram is visualized in figure 3.4.

![Feature categories as described in Mey97](image)

Figure 3.3: Feature categories as described in Mey97

The taxonemia rule in Mey97 demands from every heir to introduce a new feature, redeclare an inherited feature or add an invariant clause. The classes DEFERRED_COMMAND and DEFERRED_QUERY neither introduce new features, nor redeclare an inherited feature, nor add a new invariant clause. It is sufficient to put a flag representing the state deferred in the class COMMAND and QUERY respectively. Further, there is no need for a class COMMAND in ERL, because it would not introduce any new features.

![Feature categories](image)

Figure 3.4: Feature categories

We need the class QUERY, because a parent feature of an attribute can be an attribute or a function. Functions and attributes must therefore have a common parent class.
3.3.5 Features in ERL

The discussion from the previous section leads to the class diagram in figure 3.5 which characterizes the feature relations in ERL.

The topmost class of the feature hierarchy is `ERL_FEATURE`. It defines the reflection information common to all kinds of features.

Features inheritance

Normally, every type provides a set of features. Some of them are inherited and others are declared. Since Eiffel supports sophisticated inheritance mechanisms, such as redeclaration, we had to make the following decision: an inherited feature always has at least one parent feature which is never the same `ERL_FEATURE` instance as the reflected feature. This also applies if the feature was inherited without any redeclaration. This leads to a clear feature inheritance hierarchy and a clear relation between types and their features.

A feature can have multiple parents, if it is joining multiple deferred features. In the cases of merging and selecting, a feature does not only have multiple parents, but also one effective parent. Consider the example in figure 3.6 that shows feature merging from three parents.

![Diagram of feature merging](#)

The list of parents in this example is: `g`, `h` and `i`. The effective parent is `h`, because it is not undefined.
To query the parents and effective parent we use

\[ \text{parents: } \text{BAG}\{\text{ERL\_FEATURE}\} \]

and

\[ \text{effective\_parent: ERL\_FEATURE} \]

Pre- and postconditions

So far, Eiffel provides pre- and postconditions for routines. In the next Eiffel release they will also be available for attributes. Consequently, every feature in ERL has pre- and postconditions. An agent routine takes a tuple containing the target and actual parameters. In Eiffel, actual parameters are called operands. To keep consistency, the pre- and postcondition queries in \( \text{ERL\_FEATURE} \) also take a tuple. We use:

\[ \text{precondition (operands: TUPLE): BOOLEAN} \]

and

\[ \text{postcondition: BOOLEAN} \]

to query them.

Because features can have pre- and postconditions we need a possibility to check whether the operands are valid:

\[ \text{valid\_operands (operands: TUPLE): BOOLEAN} \]

The situation above entails another need. We can only check postconditions if the feature was invoked first. Therefore, we have to query if a postcondition can be checked:

\[ \text{can\_postcondition\_be\_checked: BOOLEAN} \]

Miscellaneous

- \textit{names: SUBSET [STRING]}. Other than a type or a class, a feature can have multiple names.
- \textit{set\_object (object: ANY)}. We want to invoke effective features. Therefore, we can set the object to operate on.
- \textit{feature\_clause: ERL\_FEATURE\_CLAUSE}. Feature clause, where the feature belongs to.
- Feature status reports:
  - \textit{is\_deferred: BOOLEAN}. Is feature deferred?
  - \textit{is\_frozen: BOOLEAN}. Is feature frozen?
  - \textit{is\_inherited: BOOLEAN}. Is feature inherited?
  - \textit{is\_object\_set: BOOLEAN}. Is object to operate on set?
  - \textit{is\_obsolete: BOOLEAN}. Is feature obsolete?
  - \textit{is\_redefined: BOOLEAN}. Is feature redefined?
  - \textit{is\_renamed: BOOLEAN}. Is feature renamed?
  - \textit{is\_selected: BOOLEAN}. Is feature selected?
State of a feature

A feature has at least one name and is assigned exactly to one feature clause. Some inheritance rules also apply: if a feature is redeclared (renamed or redefined) or selected, it is inherited. If it is inherited, the parent bag is not empty. At last, if a feature is redefined, it has no effective parent.

3.3.6 Routines

We call the general construct of procedures and functions routine. The corresponding ERL class is ERL_ROUTINE. In most cases, an ERL routine acts as a wrapper to the routine of the the agent mechanism. This approach has a drawback. A routine, in case of an agent, takes a tuple with the operands (actual arguments). The first operand is always the target of the routine. But, as we have seen above, it should be possible to set the object to operate on in advance of invoking the routine. This approach does not work with agent routines, because it is not possible to make tuples of a fixed length and to resize them respectively. Thus, we must give the target at the first position of the operand tuple.

The old construct is sometimes needed in the postcondition of a routine. It should still be possible to check the postcondition after invoking a routine dynamically. Therefore, an ERL routine caches the old values before invoking itself.

We use ERL routines almost in the same way as we use agent routines:

- \texttt{set\_operands (a\_operands: TUPLE)}. Assign the operands.
- \texttt{operands: TUPLE}. Which operands are set?
- \texttt{call}. Invoke the routine.
- \texttt{was\_call\_made: BOOLEAN}. Redeclared version of \texttt{can\_postcondition\_be\_checked}.

Formal arguments

Routines can have formal arguments. We can ask for them with

\[
\texttt{formal\_arguments: LIST [ERL\_ARGUMENT]}
\]

The offered reflection facilities of ERL\_ARGUMENT are:

- \texttt{name: STRING}. Name of the argument.
- \texttt{type: ERL\_TYPE}. Type of the argument.
- \texttt{position: INTEGER}. Position of the argument within the list of the formal arguments.
- \texttt{routine: ERL\_ROUTINE}. To which routine the argument belongs.

Procedure

ERL\_PROCEDURE is a routine with the additional

\[
\texttt{is\_creator: BOOLEAN}
\]

to ask if the procedure is listed in the create clause of its declaring type.
3.3 Design of ERL

3.3.7 Queries
A query is a feature that returns a result. The result has a value and a type. The corresponding queries in \texttt{ERL.QUERY} are

\begin{verbatim}
    item: ANY

    result_type: ERL_TYPE
\end{verbatim}

The value of the attribute must conform to its declared result type. To check this, \texttt{ERL.QUERY} provides the status report

\begin{verbatim}
    erl_result_type_conforming_to (other: ANY): BOOLEAN
\end{verbatim}

Attribute
\texttt{ERL.ATTRIBUTE} introduces only one more feature to replace the value of the reflected attribute in the assigned object.

\begin{verbatim}
    replace (a_value: ANY)
\end{verbatim}

Function
\texttt{ERL.FUNCTION} is the combination of a query and a routine.

3.3.8 Access to ERL
We have not spoken about how we can get the type of an object and any other system information. It is desirable, if the main class \texttt{ANY} includes a query to get its type. Changing \texttt{ANY} is certainly possible, but as long as the changes are not integrated in an Eiffel release, we need another way to get access to reflection.

Further, \texttt{ANY} is not the best place to provide queries to ask for information about a whole system, such as all cluster, all types and all objects. A more universal access point would top ERL off.

For this two reasons, ERL provides a class called \texttt{ERL.UNIVERSE} to give a global entrance to the Eiffel reflection. \texttt{ERL.UNIVERSE} offers the following functionality:

- \texttt{clusters: TREE [ERL.CLUSTER]}. All clusters of the running system.
- \texttt{objects: SUBSET [ANY]}. All objects in the running system.
- \texttt{types: SUBSET [ERL.TYPE]}. All used types in the running system.
- \texttt{exists_type_for_name (a_name: STRING): BOOLEAN}. Does type with \texttt{a_name} exist?
- \texttt{object_type (a_object: ANY): ERL_TYPE}. Type of the given object \texttt{a_object}.
- \texttt{type_for_name (a_name: STRING): ERL_TYPE}. Type for the given name \texttt{a_name}.
- \texttt{class_for_name (a_name: STRING): ERL_CLASS}. Class for the given name \texttt{a_name}.

The list above reflects the minimal set of table \ref{tab:ERL.UNIVERSE}. As with feature searching, it is convenient to have queries to search clusters and types by name.

- \texttt{type_by_name: ERL_TYPE}. Search type by name.
- \texttt{cluster_by_name: ERL.CLUSTER}. Search cluster by name.
3.4 Partial implementation of ERL

In the case where a library depends on a particular compiler and runtime system, we should separate its implementation from its interface to simplify the replacement of various implementations.

There are two common ways to separate an implementation from its interface. First, the bridge pattern introduced in [HJVG95] suggests to separate the implementation and its interface in two class hierarchies with the same structure. Then, the interface is in a has-a relation to its implementation (see figure 3.7 left).

![Figure 3.7: Bridge pattern; implementation by inheritance](image)

The second approach is implementation by inheritance. An implementation class implements the features of its deferred parent class (see figure 3.7 right).

We have chosen implementation by inheritance, because of the following reasons:

- It is more intuitive to let the interface deferred and provide the implementation by implementing the deferred features.
- The interface has nothing to know about its implementation. Thus, the separation is more strict.
- Replacing an implementation in the bridge pattern, as suggested in [HJVG95], is more difficult.

The third reason is not so intuitive at first glance. If the classes of a new implementation are named different, we must change more code than with implementation by inheritance. In ERL, for example, no class, except ERL\_UNIVERSE, contains a declaration of a type ending with IMP.

To overcome this problem, we could make a common interface connecting the implementation and its interface. A similar approach is used in EiffelVision2. Nevertheless, we kept at our decision.

We discuss the partial implementation of ERL in the remainder of this section. We have composed the name of an implementing class with the name of the interface and the postfix IMP. We have put particular parts of the implementation into the interface, because they should remain the same for every special implementation of ERL.

The Eiffel runtime system does not support full reflection yet. Some reflection information are available via an instance of INTERNAL, but still not all of the suggested facilities in ERL. Our partial ERL implementation is restricted to the support from INTERNAL. In this section we discuss this partial implementation. If we state that we were unable to implement a feature we mean the corresponding support is missing in INTERNAL and the Eiffel runtime system respectively.

**ERL\_CLUSTER, ERL\_FEATURE\_CLAUSE, ERL\_GENERIC** and **ERL\_CONSTRAINT** have no implementation, because the Eiffel runtime system does not provide information about clusters, feature clauses, and formal generic parameters yet.
3.4 Partial implementation of ERL

3.4.1 Universe

We get access to ERL over an instance of \texttt{ERL\_UNIVERSE}.

- \texttt{exists\_type\_for\_name} 
  This query checks if \texttt{INTERNAL} knows the type id for the given string.

- \texttt{class\_for\_name, object\_type, type\_for\_name} 
  These queries return the demanded information as they are available in ERL.

- \texttt{cluster\_by\_name, clusters, objects, type\_by\_name, types} 
  We have not implemented these queries, because the corresponding support is still missing in Eiffel.

3.4.2 Type

The Eiffel Kernel Library class \texttt{INTERNAL} provides several queries to obtain information about types and attributes. We use them in \texttt{ERL\_TYPE\_IMP}.

Initialization

There are two ways of initializing an \texttt{ERL\_TYPE\_IMP} instance.

- \textit{make (a\_object: ANY)} initializes a type with a particular object. We retrieve the name of the type from \texttt{INTERNAL}. The type must not be deferred.

- \textit{make\_for\_name (a\_name: STRING)} initializes a type with the name \texttt{a\_name}. This creator can be used with deferred types, where no object can exist.

Type reflection

- \texttt{actual\_generics} 
  According to a bug in \texttt{INTERNAL}, we must determine the number of generic parameter by counting the commas in the type name. But not all commas are of interest because it is possible to declare types with nested generic types:

  \[
  \texttt{A\_CLASS [STRING, PROCEDURE [ANY, TUPLE]]}
  \]

  Therefore, we ignore commas in square bracket pairs within the outermost pair.

- \texttt{base\_class} 
  Since \texttt{INTERNAL} provides a query to retrieve the class name of a type, we can get the base class of a type (see \texttt{3.4.3} also).

\footnote{\texttt{generic\_parameter\_count} works only with \texttt{TUPLE} types}
Chapter 3. Reflection for Eiffel

We are able to check conformance between two objects. Since we are not yet able to check conformance between two ERL types directly, we must do the check on the objects that initialized the types. Because we are only able to check conformance between two objects, we cannot check conformance between two deferred types.

The corresponding support is still missing in Eiffel. The implementation generates a dummy feature clause with the client \texttt{NONE} to fulfill the invariants of a type.

This feature looks for the searched name in the name set of each feature.

The set of all features of a type are merged from the features of each feature clause. We cache the merged set to perform faster search operations. We compose the primary groups of features introduced in section 3.3.2 (p. 32) by looping over the set of features. After we have made a particular group for the first time, we cache it. \texttt{ERL\_TYPE} is already implementing these operations because the data structures are declared in the interface. So far, \texttt{INTERNAL} only supports retrieving the attributes of a type. Therefore, we redefine features in \texttt{ERL\_TYPE\_IMP} to return the set of all attributes. It is important to know that attribute retrieving only works if we initialize the type with an object. According to the reasons above, we were not able to implement the features \texttt{deferred\_features} and \texttt{routines}.

A type has actual generic if the type string contains at least one left square bracket.

The corresponding support is still missing in Eiffel. A possible solution is to provide an agent. It takes an object of the reflected type and returns the result of the check whether the object satisfies the invariants specified by the reflected type.

We are only able to retrieve information whether a type is basic or expanded for attributes. This support is still missing in Eiffel for all other kinds of features.

Since testing whether a type is deferred is not yet implemented, its solution in ERL is a hack. We test if we can instantiate an object of a given type string. If the runtime environment of Eiffel throws an exception, the type is deferred.

The corresponding support is still missing in Eiffel.

The corresponding support is still missing in Eiffel. A possible solution is to lookup the feature call in the vtable.

If the reflected type is not deferred, we can generate a new uninitialized instance of it. \texttt{INTERNAL} provides this feature.
The corresponding support is still missing in Eiffel. The parents bag contains ANY to fulfill the invariants of a type.

**Problems with INTERNAL**

**Object dependence** As described above, reflection information retrieving is only possible, if we initialize a type with an object, because INTERNAL only works with objects. This is not sufficient, because a type is independent from a particular object. If a forthcoming Eiffel release will support this issue, we must adjust the ERL implementation.

**Basic types** It is not possible to determine the correct type of a basic type object. The reason is the call semantics of a routine in Eiffel. If a routine expect an argument of type ANY, Eiffel converts the basic type to its reference type.

We cannot get an instance of a basic type. This is correct, because we must not instantiate expanded types. But at least, the new_instance query should return an object that has the reference type of the expanded type. A forthcoming Eiffel release should fix this issue.

**Expanded types** It is possible to determine the type of an object of an expanded type, if it has not a reference type. But we cannot instantiate it. We should be able to get an object of an expanded type anyhow.

If an expanded type acts as an actual generic, INTERNAL returns a wrong type string. This issue does not take place with basic types. Example:


definable B

class A [G]

The declared type A [B] gets the type name A [expanded B] from INTERNAL. But if we demand the type id of this name with

\[ a: A [B] \]
\[ str: STRING \]
\[ internal: INTERNAL \]
\[ type_id: INTEGER \]

**create internal**

\[ str := internal.type_name(a) \]
\[ -- 'str' has value "A [expanded B]" \]

\[ type_id := internal.dynamic_type_from_string(str) \]
\[ -- 'type_id' has value -1 \]

INTERNAL returns -1 to state that this type does not exist.
ANY

The type id returned from INTERNAL of an object of type ANY or of the string "ANY" is 0 to indicate that the demanded type is NONE. This is incorrect.

3.4.3 Class

Eiffel does not yet support a query whether a class is generic. Therefore, the creator of a class takes not only its name but also a flag indicating if the class is generic or not.

\[
\text{make (a\_name: STRING; generic: BOOLEAN)}
\]

Since we instantiate mostly a class from a type, the name is available from INTERNAL and the generic flag from has\_actual\_generics. We can also instantiate a class in ERL\_UNIVERSE, but maybe we do not know if the desired class is generic. This is non-satisfying, but we are unable to solve it better so far.

Deriving a type

We are able to derive types which the runtime system knows. It is important to say that we can only derive types known to the runtime system, otherwise the system will throw an exception. Therefore, ERL\_CLASS provides a query to check if a type with a given list of actual generics is derivable from the reflected class.

An actual generic parameter must fulfill the constraints of its corresponding formal generic parameter. It is possible to reduce this verification to the check whether the number of given actual generics is equal to the number of formal generics. We are in fact able to generate a type not fulfilling its generic constraints. However, we cannot use a corresponding instance, because a correct assignment attempt is impossible. Consider the following example:

\[
\text{class A [G, H - COMPARABLE]}
\]

We try to create a new instance of type A [STRING, TUPLE]. The obtained object cannot be assigned to a variable of type A [STRING, TUPLE], because the compiler does not allow such a variable declaration.

The query type\_with\_actuals constructs the type string with the class name and the list of the given actual generics. Then it creates a new ERL\_TYPE\_IMP instance.

Class reflection

\[
\text{cluster}
\]

The corresponding support is still missing in Eiffel. A dummy cluster is made to fulfill the invariants of a class.

\[
\text{exists\_type\_with\_actuals}
\]

It constructs the type string with the class name and the list of the given actual generics. Then it checks if INTERNAL knows a type id for the constructed string.

\[
\text{formal\_generics}
\]

The corresponding support is still missing in Eiffel. If the class is generic, the formal generic parameter list contains a dummy parameter without any constraint to fulfill the invariants of a class.
### 3.4 Features

A feature can have multiple names and belongs always to a feature clause. Therefore, the creator of a feature takes a set of names and its feature clause:

\[
\text{make (a_names: SUBSET [STRING]; a_feature_clause: ERL_FEATURE_CLAUSE)}
\]

We invoke features on particulars object. These object must conform to the reflected type of feature to invoke. *ERL_FEATURE* provides the corresponding check:

\[
erl\text{type_conforming_to (other: ANY): BOOLEAN}
\]

This query calls *erl_conforms_to* of *ERL_TYPE*. Actually, we can make this directly, but since a *create* statement is not allowed in a precondition, we must wrap the call into *ERL_UNIVERSE* to get the type of *other*. We hope *ANY* will provide a query to get the ERL type of an object to solve this issue.

#### Feature reflection

We do not list all facilities of *ERL_FEATURE_IMP* here, because we were not able to implement most of them. The same applies to the following discussion of routines, queries, functions and attributes.

#### Routine

Most parts of an ERL routine are wrappers to an agent routine. Therefore, we only list the features of special interest.

- **call**
  
  Before we make a call, we must save the values used by possible *old* statements in the postcondition of the routine. The corresponding support is still missing in Eiffel.

- **formal_arguments**
  
  Determining the types of the formal arguments is cumbersome. For every argument, we have to parse the type character. On the basis of this character we can find out, if the type is a reference type or one of the basic types. We always try to create an ERL type with a corresponding object. Therefore, we insert instances of reference types which are not deferred into the list. If the type of the formal argument is a basic type, we use its reference type to store it in the list.

#### Argument

ERL provides the reflection information name, type, position, and routine for a formal argument (see 3.3.6). We assign these information when we create an instance of *ERL_ARGUMENT_IMP*:

\[
\text{make (a_name: STRING; a_type: ERL_TYPE;}
\]

\[
a\text{position: INTEGER; a_routine: ERL_ROUTINE)}
\]

#### Query

To check if the value of the attribute conforms to the result type of the query, *ERL_QUERY* provides the function:

\[
erl\text{result_type_conforming_to (other: ANY): BOOLEAN}
\]
The implementation of this query has two problems. Sometimes, INTERNAL handles basic type result objects with their reference types. Normally, the declared result type is a parent type or the same type of the attached object type. With basic types it is the other way round. The declared result type is the expanded version of its reference type. Therefore, we must first check if the result type is a basic type.

Second, since we cannot check type conformance with deferred types, because we have no object to make the check, we must pass the test.

Function

The last actual generic parameter of an agent function is the result type of the function. But as we are only able to get the reference type of a basic type, the ERL type of the result type is not always correct.

Attribute

Reflection on attributes is the best reflection support in Eiffel so far. The class INTERNAL provides the corresponding facilities.

INTERNAL numbers the attributes of a type. We retrieve an attribute of a particular object with this internal number. The numbering does not always reflect the same position of an attribute as it occurs in the type.

attribute_type

We must perform three steps to get the ERL type of a particular attribute. First, we must determine the type id of the reflected object. Second, we have to demand the static field type of the attribute at the particular position. And third, we must create a new ERL type with an instance of the attribute type or with its name, if it is deferred.

```
attribute_type

field_type, o_type: INTEGER
type: ERL_TYPE_IMP
internal: INTERNAL

create internal
o_type := internal.dynamic_type (object)
field_type := internal.field_static_type_of_type (position, o_type)
if is_deferred_type (field_type) then
  create type.make_for_name (internal.type_name_of_type (field_type))
else
  create type.make (internal.new_instance_of (field_type))
end
```

names

An attribute can only have one name, but in general features can have multiple names. Eiffel must fix this issue in forthcoming releases.

replace

INTERNAL uses different setters for attributes of a basic type. Therefore, the replace operation determines which type the attribute has to invoke the correct setter.

A setter for general expanded types does not exist in INTERNAL.
3.5 Conclusion

To design the Eiffel Reflection Library we used the approach of mapping the object model of Eiffel as exhaustive as possible but with respect to usability and taxonomy. The achieved result fulfills these requirements. ERL reflects the object-oriented concepts introduced in [Mey97] more precisely than the reflection libraries discussed in chapter 2. Moreover, ERL fulfills the expectations we proposed at the beginning of this chapter (section 3.2).

One of the specialties of ERL is its ability to build inheritance structures of features. During runtime we can exactly determine how features are adapted in a class hierarchy. This functionality is new in the field of reflection.

The use of ERL is simple and intuitive. Here is a small example: assume we have a class LINKED_LIST [G] and a generic derivation list: LINKED_LIST [STRING]. Assume further, we want to get the actual and formal generic parameter of list and LIST respectively:

```
list: LINKED_LIST [STRING]
type: ERL_TYPE
list_class: ERL_CLASS
universe: ERL_UNIVERSE
actuals: LIST [ERL_TYPE]
formals: LIST [ERL_GENERIC]
```

```
create universe
create list

-- actual generic parameter
type := universe.object_type (list)
actuals := type.actual_generic

-- formal generic parameter
list_class := type.base_class
formals := list_class.formal_generic
```

If object_type was already declared in ANY we would not need to get the type of list from universe.

The same situation is not that intuitive in .NET.

```
List<string> list = new List<string>();
Type type = list.GetType();

// actual generic parameter
Type [] actuals = type.GetGenericParameters();

// formal generic parameter
Type claz = type.GetGenericTypeDefinition();
Type [] formals = claz.GetGenericParameters();
```

.NET does not distinguish between the type List<string> and the class List<T>. Both are instances of Type. This implicates that retrieving formal and actual generic parameters are named identically: GetGenericParameters.
Partial ERL implementation

The implementation of ERL is restricted to the scope of functionality of `INTERNAL`. In appendix B we have listed which features of ERL are implemented, partially implemented or not implemented. In that table we can see that Eiffel has no support for `ERL_CLUSTER`, `ERL GENERIC`, `ERL CONSTRAINT`, and `ERL FEATURE_CLAUSE`.

Next, we summarize why some of ERL’s features are only partially implemented (table 3.7) and which problems and bugs we have found while implementing ERL.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Feature</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERL_UNIVERSE</td>
<td><code>cluster_by_name, type_by_name</code></td>
<td>Implemented, but not yet the facility to fill the data structures.</td>
</tr>
<tr>
<td></td>
<td><code>object_type, type_for_name</code></td>
<td>We cannot use basic type. Eiffel does not use expanded type in a generic derivation correctly.</td>
</tr>
<tr>
<td>ERL_TYPE</td>
<td><code>actual_generics</code></td>
<td>We must count the number of generic parameters by hand. We cannot retrieve basic actual parameter types.</td>
</tr>
<tr>
<td></td>
<td><code>deferred_features, features, routines</code></td>
<td>Implemented, but since we can only reflect attributes, these information are not available.</td>
</tr>
<tr>
<td></td>
<td><code>feature_by_name</code></td>
<td>Only attributes.</td>
</tr>
<tr>
<td>ERL_CLASS</td>
<td><code>conform_generic_constrains</code></td>
<td>Implemented, but since we cannot obtain the formal generic parameter list the number of formal generic parameters is unknown.</td>
</tr>
<tr>
<td></td>
<td><code>is_generic</code></td>
<td>So far, the class gets this information when creating the base class within the type. But if we want to create the class from scratch, the information is not yet available.</td>
</tr>
<tr>
<td></td>
<td><code>type_with_actuals</code></td>
<td>It is not possible to derive new user defined types unknown to the system or which have basic actual parameter types.</td>
</tr>
<tr>
<td>ERL_FEATURE</td>
<td><code>names</code></td>
<td>Attributes have only one name.</td>
</tr>
<tr>
<td></td>
<td><code>valid_operands</code></td>
<td>Only available in agent routines.</td>
</tr>
<tr>
<td>ERL_ROUTINE</td>
<td><code>formal_arguments</code></td>
<td>Eiffel wraps basic types to their reference type.</td>
</tr>
<tr>
<td>ERL_FUNCTION</td>
<td><code>result_type</code></td>
<td>Eiffel wraps basic types to their reference type.</td>
</tr>
<tr>
<td>ERLATTRIBUTES</td>
<td><code>replace</code></td>
<td>It is not possible to replace the value of an attribute of expanded type.</td>
</tr>
</tbody>
</table>
Problems and bugs concerning *INTERNAL*

- Type reflection depends on a particular object.
- Eiffel wraps basic type to their reference type.
- It is not possible to generate new instances of basic types.
- The type of an actual generic parameter is *NONE*, if it is of a basic type.
- The use of expanded types in the list of actual generic parameters leads to wrong type names.
- Generic parameter count works only for tuples.
- The type id of *ANY* is 0, the same as for *NONE*.
- It is not possible to derivate generic types unknown to the system.
- We cannot replace values of attributes of expanded types.

Changed and added features in *ROUTINE*

- *open_operand_type* is exported to *ERL_ROUTINE_IMP*.
- The export status of *runtime constants* is set to *ERL_ROUTINE_IMP*.
- We have added *formal_argument_type_string* under *runtime constants*. 
Chapter 4

ObjectBrowser

4.1 Introduction

The ObjectBrowser is an EiffelVision2 application to browse through object structures at runtime. Browsing object structures is not only useful to debug programs but also make complex structures of programs visible during runtime. The latter issue addresses particularly beginners in object-oriented programming to get a deeper understanding of their own programs. Moreover, the ObjectBrowser is an appropriate use case of the Eiffel Reflection Library (ERL).

The implementation of ObjectBrowser contains two specialties. First, it uses dot from GraphViz [GN96] to lay out the object structure. Second, to create a meaningful graph for the layout, it uses the graph library designed by Olivier Jeger. At the time of the implementation, the graph library is a part of Olivier’s ongoing master thesis [Jeg04].

In this chapter we discuss the implementation of the ObjectBrowser and describe how it can be used.

4.2 Requirements and functionality

We have made some minimal requirements on the ObjectBrowser when this master thesis started:

- We should be able to traverse a given object structure and show details about each object.
- We should be able to change the values of basic type attributes of an object, such as INTEGER.
- The ObjectBrowser must be easy to use, i.e. the alignment of the GUI parts has to be intuitive and we must describe how the provided operations can be executed.

Traverse a given object structure means when starting at a root object, we should be able to retrieve information of its attributes and their attributes etc. It is most reasonable to present this in a tree structure, where the attributes are the nodes. Such a structure is known from the debugging mode of EiffelStudio. The displayed details of each attribute should contain information available from the reflection library.

In the meanwhile the resulted functionality of the ObjectBrowser exceeded the minimal requirements. It is not only possible to traverse the given object structure in tree mode, but also to visualize it as a graph with a given depth.
Chapter 4. ObjectBrowser

If a compiler for a particular programming language is available for various operating systems, platform independence in its application is very important. Therefore, we implemented the ObjectBrowser with EiffelVision2. We retrieve the visualized reflection data from ERL (see chapter 3).

We will now have a look at the application and its user interface.

The main window of the ObjectBrowser is divided into three parts (see figure 4.1):

1. **Tool bar.** The ObjectBrowser provides the operations arranged in a tool bar, it includes from left to right:
   - Quit the ObjectBrowser.
   - Graph view depth setting.
   - Show graph view with desired depth.
   - Make selected node to root node of tree; a new view will be generated. The old view is available through the back button.
   - History back and forward.
   - Refresh view.

2. **Object tree.** In the left hand side of the window is a tree structure. A node in the tree is an attribute belonging to the type represented by its parent node. An attribute has a name and a declared type. The root node is the type of the object with which we started the ObjectBrowser. The icon at the beginning of each node describes if a type is a reference type, an expanded type, a basic type or void. To prevent confusion, they are the same icons as in EiffelStudio.

3. **Object information.** In the right hand side of the window are some multicoloumn lists describing known reflection data of the marked attribute in the object tree. The red flash indicates unavailable information. The reflection data includes:
   - **Type.** This panel shows available information about the type of the selected object, including its name, base class name, value and if it is a basic type, deferred and/or expanded. Additionally, it shows the type names of its actual generic parameter(s), if the object type is generic.
   - **Class.** This panel shows available information about the base class of the type to its left. At the time, only the class name is available.
   - **Routines.** This panel would list all routines and deferred features of the reflected type, if the Eiffel runtime supported it.
   - **Attributes.** If the reflected type is a reference or expanded type the panel lists its attributes. A list row contains the name of the attribute, the type of the attached object and its value. The attached type can be a descendant of the declared attribute type in the object tree.
     - If the reflected type is a basic type, the panel does not show its attributes, but its value.
     - Then we can additionally replace the value of the object.

We lay out the graph views of object structures with *dot*, a tool from GraphViz. The graph view starts with the root node at the left and grows with each step of the depth selected by the user to the right. Figure 4.2 visualizes the graph view of the object structure displayed in figure 4.1 with depth three.
## 4.2 Requirements and functionality

![ObjectBrowser](image)

**Figure 4.1:** *ObjectBrowser*

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>LINKED_LIST [STRING]</td>
</tr>
<tr>
<td>base class</td>
<td>LINKED_LIST</td>
</tr>
<tr>
<td>is basic type</td>
<td>False</td>
</tr>
<tr>
<td>is defined</td>
<td>False</td>
</tr>
<tr>
<td>is expanded</td>
<td>False</td>
</tr>
<tr>
<td>is obsolete</td>
<td>False</td>
</tr>
<tr>
<td>has actual generics</td>
<td>True</td>
</tr>
<tr>
<td>actual generic 1</td>
<td>STRING</td>
</tr>
</tbody>
</table>

**Figure 4.2:** *Graph view of object structure from figure 4.1*

![Graph view](image)

**Figure 4.2:** *Graph view of object structure from figure 4.1*
Chapter 4. ObjectBrowser

4.3 Implementation

In this section we discuss some aspects of the ObjectBrowser’s implementation. Mainly the part of generating the graph view of the object structure.

4.3.1 Implementation of the GUI

The main window has two components, the tool bar and the browse panel. The tool bar provides the main operations of the ObjectBrowser. The browse panel is divided into the object tree and object information panel.

Object tree

The ObjectBrowser is started with a root object to show its reflection data. At the beginning the root node of the object tree is the root object. Its children are the attributes of the root object’s type. In general, the ObjectBrowser only constructs and shows the children of a node if we have selected the node. This approach has two intuitive reasons. First, a provided object structure could contain cycles and therefore, the object tree would grow indefinitely if the ObjectBrowser tried to construct the whole tree. Second, even if we are able find cycles in graphs, we rarely need the whole object structure.

The object tree is actually an EV_TREE. The only specialty are its nodes. They are an extension of EV_TREE_ITEM. The new nodes of type OBJECT_TREE_ITEM contain the displayed attribute. Therefore, it is not needed to search the corresponding attribute in the type of its parent node if we select a node in the tree.

Object information

We use an EV_MULTI_COLUMN_LIST to display the information about the type, class and attributes of an object. If the type of the selected attribute in the tree is a reference or expanded type, the list at the bottom of the object information panel contains its attribute. The type of an attribute in the list may differ from the type of the same attribute in the object tree, because the list shows the types of the attached objects.

Navigation

An object of type BROWSER generates and lays out the panels beneath the tool bar. That means, it is responsible for the reflection information retrieving from ERL and for displaying these information in the panels.

The panels are partly combined in other components, e.g. FRAMED_MULTI_LIST. The graph view of an object structure starts at the root node of the tree. Therefore we provide the facility to make a selected node to the new root node. For that purpose we generate and initialize a new instance of type BROWSER with the new root node. Because it is desirable to keep the old browser content, we have implemented the browsing functionality back and forward. The ObjectBrowser uses two LINKED_STACK [BROWSER] to store the forward and back contents.

It would also be possible to store only the data of the panels, but this is more complicated to implement and would cause a computation overhead. The used approach to create a new content of the window is much easier to realize, but has additional memory overhead because all widgets must be saved.
4.3 Implementation

4.3.2 Object structure layout

We decided to display the object structure in the shape of a graph. To generate the view, we make two steps. The first is to generate a graph of the object structure from the object tree with a certain tree-depth. The second step is to lay out and visualize the graph. We make the graph with Olivier Jeger’s graph library \cite{Jeg04}. We lay out and visualize the layout with the \textit{dot} application of GraphViz \cite{GN96}.

Graph library

It is very simple to build a graph with Olivier’s graph library. We can decide between combinations of directed or undirected and weighted or unweighted graphs. With the creation procedures we tell the graph if it is a simple or multi graph. For each of the combinations exists two implementation variation: adjacency matrix graph and linked graph.

Every graph variation takes two generic arguments. The first is the node type, the second the edge label type.

We have chosen to take a \texttt{LINKED\_GRAPH [STRING, STRING]} for our purpose, since we deal only with strings. The linked graph is unweighted and directed. It is initialized as multi graph.

To build the graph, we walk through the object tree and put each node of the tree as node of the graph. Same nodes appear in the graph only once. For each child of a node we add an edge between the parent and the child. The provided depth by the user indicates how many depth levels of the tree the graph generator should add to the graph. For instance, the depth \textit{three} means to put all attributes of the root object and their attributes into the graph.

GraphViz and DOT

AT&T Labs - Research has invented GraphViz \cite{GN96}. It specifies the \textit{DOT language} to describe graphs. We describe a simple graph in the DOT language as follows:

\begin{verbatim}
digraph G {
    "A" -> {"B", "C"}
}
\end{verbatim}

The directed graph \texttt{digraph G} consists of three nodes with labels "A", "B", and "C". There are two directed edges. The first between the nodes labeled "A" and "B" and the second between the nodes labeled "A" and "C".

Dot will lay out the graph as shown in figure \ref{fig:dot-layout}.

![Figure 4.3: A simple graph layouted with dot](image)

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{dot-layout.png}
\caption{A simple graph layouted with \textit{dot}}
\end{figure}
Additionally, we can define some properties of the graph, e.g. the shape of the nodes and edges, the fill color, the node and edge label font, its color, etc.

**Graph layout generation**

The class `GRAPH_VISUALIZER` takes care of the DOT output generation of a given graph. We can specify some selected options of the graph:

<table>
<thead>
<tr>
<th>Option</th>
<th>Initialized</th>
<th>Queried</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge color</td>
<td>black</td>
<td><code>edge_color</code></td>
<td><code>set_edge_color</code></td>
</tr>
<tr>
<td>edge label font color</td>
<td>black</td>
<td><code>edge_label_font_color</code></td>
<td><code>set_edge_label_font_color</code></td>
</tr>
<tr>
<td>edge label font bold</td>
<td>False</td>
<td><code>is_edge_label_font_bold</code></td>
<td><code>enable_bold_edge_label_font</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>disable_bold_edge_label_font</code></td>
</tr>
<tr>
<td>edge label font italic</td>
<td>False</td>
<td><code>is_edge_label_font_italic</code></td>
<td><code>enable_italic_edge_label_font</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>disable_italic_edge_label_font</code></td>
</tr>
<tr>
<td>edge label font size</td>
<td>10</td>
<td><code>edge_label_font_size</code></td>
<td><code>set_edge_label_font_size</code></td>
</tr>
<tr>
<td>edge shape</td>
<td>normal</td>
<td><code>edge_shape</code></td>
<td><code>set_edge_shape</code></td>
</tr>
<tr>
<td>node color</td>
<td>black</td>
<td><code>node_color</code></td>
<td><code>set_node_color</code></td>
</tr>
<tr>
<td>node fill color</td>
<td>white</td>
<td><code>node_fill_color</code></td>
<td><code>set_node_fill_color</code></td>
</tr>
<tr>
<td>node label font color</td>
<td>black</td>
<td><code>node_label_font_color</code></td>
<td><code>set_node_label_font_color</code></td>
</tr>
<tr>
<td>node label font bold</td>
<td>False</td>
<td><code>is_node_label_font_bold</code></td>
<td><code>enable_bold_node_label_font</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>disable_bold_node_label_font</code></td>
</tr>
<tr>
<td>node label font italic</td>
<td>False</td>
<td><code>is_node_label_font_italic</code></td>
<td><code>enable_italic_node_label_font</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>disable_italic_node_label_font</code></td>
</tr>
<tr>
<td>node label font size</td>
<td>10</td>
<td><code>node_label_font_size</code></td>
<td><code>set_node_label_font_size</code></td>
</tr>
<tr>
<td>node shape</td>
<td>ellipse</td>
<td><code>node_shape</code></td>
<td><code>set_node_shape</code></td>
</tr>
<tr>
<td>left to right direction</td>
<td>False</td>
<td><code>is_rankdir_ltr</code></td>
<td><code>enable_rankdir_ltr</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>disable_rankdir_ltr</code></td>
</tr>
</tbody>
</table>

At last, we tell the graph visualizer whether we want to put the DOT output into a string or a file:

```
out: STRING
to_file (a_file_name: FILE_NAME)
```

With the used graph library it is simple to traverse the graph. We can choose between breath or depth first traversal of the graph. It does not matter in which manner we walk through the graph to generate the DOT output; we have chosen the breath first variant.
Next follows a short outline of the DOT output generation algorithm:

1. Start the graph walker at the root node of the graph.
2. Walk through the graph and do the following with each node:
   - For every outgoing edge of the visited node print:
     
     "<current node>" -> "<target node>" [label = "<edge label>"];

4.3.3 Platform independence

The ObjectBrowser uses some icons in its GUI. Therefore, we must construct the file paths to them in a platform independent way, because Windows and UNIX operating systems have different file separators ("\" vs. "/"). We use the Eiffel classes FILE_NAME and DIRECTORY_NAME to construct file and directory names respectively. The class IMAGE_LOCATOR in the ObjectBrowser project then provides an interface to the desired icons. The only thing the ObjectBrowser must know from the user is the path to its home directory. The ObjectBrowser can query it through environment variable OBJECT_BROWSER_HOME.
4.4 User guide

4.4.1 Requirements

We have tested the ObjectBrowser on Linux and Windows XP. Thus, it runs also on UNIX like platforms. The ObjectBrowser needs the GraphViz library with the tool dot. You can download it from [ ]

4.4.2 Installation

To install the ObjectBrowser in your existing project, you need to follow the instruction below:

1. Download the file ObjectBrowser.tar.gz from fluri.computerscience.ch/projects/. The file contains the directory object_browser with three subdirectories:
   - images: Icons used by the ObjectBrowser.
   - src/erl: A complete copy of the Eiffel Reflection Library.
   - src/graph_library: A complete copy of Olivier Jeger’s Graph Library.
   - src/object_browser: Source code of the ObjectBrowser.

2. Unzip the file to a directory of your choice.

3. Set the $OBJECT_BROWSER_HOME environment variable to point to the ObjectBrowser directory.

4. You should also add the directory where the dot application is located to the PATH environment variable.

5. You have two possibilities to use the ObjectBrowser:
   - Create an new project in EiffelStudio and use the provided ace file $OBJECT_BROWSER_HOME/src/object_browser/object_browser_unix.ace for UNIX like platforms or $OBJECT_BROWSER_HOME/src/object_browser/object_browser_windows.ace for Windows platforms. You can then compile and run the test case.
   - Include ObjectBrowser into an existing project:
     - Make a backup of your existing Ace file.
     - Use the provided Ace file (overwrite your existing Ace file, but not the backup): $OBJECT_BROWSER_HOME/src/object_browser/object_browser_unix.ace for UNIX like platforms or $OBJECT_BROWSER_HOME/src/object_browser/object_browser_windows.ace for Windows platforms.
     - In the selected Ace file replace the root clause test_object_browser: make with your root class and application launch procedure.
     - Add clusters that you use and are not yet listed in the Ace file.

6. The ObjectBrowser expects an object to start visualizing an object structure. It uses this object as root object of the object structure. You can use the following code to start the object browser. The argument a_root_object is the object you want the ObjectBrowser to visualize.

```eiffel
a_root_object -> show
```
4.4 User guide

browse_my_object_structure (a_root_object: ANY) is
  -- Show object structure starting at 'a_root_object'
  -- in an ObjectBrowser window.
require
  a_root_object.exists: a_root_object /= Void
local
  l_object_browser: OBJECT_BROWSER
do
  create l_object_browser
  l_object_browser.show (a_root_object)
end

ATTENTION: you must not use a precompiled library.

4.4.3 How to use ObjectBrowser

After you have successfully compiled your project, you can execute it. You should see an Object-
Browser window such as shown in figure 4.4. It was initialized with the following code:

browse_a_list_structure is
  -- Browse a list structure in ObjectBrowser.
local
  ob: OBJECT_BROWSER
  i: INTEGER
  str: STRING
  list: LINKED_LIST [STRING]
doi
  create ob
  -- create a list with 100 elements
  create list.make
  from
    i := 1
  until
    i = 100
  loop
    str := "item"
    str.append (i.out)
    list.extend (str)
    i := i + 1
  end
  ob.show (list)
end

The ObjectBrowser window is divided into three parts: the object tree, the object information
panel and the tool bar. Next, we will see how to use these three parts.
Object tree

The root node of the object tree has the type of the root object as title. The root object is the object that you have provided to launch the ObjectBrowser with

\[ \texttt{ObjectBrowser.show (a\_root\_object)} \]

Every tree node, except the root node, represents an attribute. Because of that we will use the word attribute, when speaking of tree nodes.

At the start point of the ObjectBrowser, the object tree shows only the root node and its children, hence the attribute of the root object’s type. As you can see, the attributes have different icons at the left. They tell you which kind of object is attached to the attribute. The meanings of those icons are visualized in figure 4.5

- Object
- Expanded object
- Void
- Object of basic type

**Figure 4.5: Kind of nodes in the object tree**

If you click on an attribute, the attributes of its type become expanded. The ObjectBrowser does not construct a subtree of attributes until you click on an attribute.
Object information panel

The object information panel is divided into four panels. The red flash icon means that the corresponding information is not available yet. You can read the reason for that in section 3.4. We will now look at the four panels:

- **Type.** The type panel shows information available of the attribute type.
- **Class.** The class panel shows information available of the attribute type’s base class.
- **Routines.** The routines panel shows information available of the routines and deferred features of the attribute type.
- **Attributes or Value.** The bottom panel of the object information panel depends on the kind of attached object to the attribute. If the type of the attached object is a basic type, you have the possibility to replace its value. Otherwise the attributes of the selected attribute in the tree are shown. In these cases, the middle column of the list tells you of which type the attribute’s attached object is. It is possible that the declared attribute type (in tree) is a parent type of the attached object’s type. As an example see figure 4.6. The declared type of the attribute is `LIST [STRING]`, but the attached object is of type `LINKED_LIST [STRING]`.

![Figure 4.6: Declared type vs. attached type of an attribute](image)

Tool bar

In figure 4.7 you can see the tool bar along with the meaning of each command group. We look at the meaning of these command groups now:

- **Quit.** This button just quits the ObjectBrowser. It has the same effect as a click on the cross on the top right of the window.
- **Graph layout of object structure.** If you press the blue arrow button, the ObjectBrowser will lay out the browsed object structure in a graph view. The depth indicates how deep the ObjectBrowser should walk through the tree. Depth one means only the root object, depth
two means the root object and its attributes, depth three means the root object, its attributes and their attributes, and so on. In figure 4.8 you can see the generated graph layout with depth four of the object structure from figure 4.4.

- **Make selected object to root object.** This button makes the selected attribute in the tree to the root node. The ObjectBrowser replaces the old browse content by new browser panel. The old browse content is available through the back button.

- **Back & Forward.** We use the back and forward buttons exactly as in an Internet browser. If you have clicked on the make selected object to root object-button, you can get the old browse content by pressing the back button. If you have clicked on the back button, you can get the old browse content by pressing the forward button.

- **Refresh.** If you press this button, the ObjectBrowser refreshes the whole browser content. You may use this if you have changed a value of an attribute to update the value line in the type panel.

**Figure 4.7:** The ObjectBrowser’s tool bar

**Figure 4.8:** Graph layout of object structure from figure 4.4 with depth 4.
Chapter 5

Conclusion

5.1 Summary

The intention of this work was to design a reflection library for Eiffel. To fulfill this task, we made the following steps:

1. We began with an evaluation of the existing reflection APIs of Java, the .NET Framework, Smalltalk and Ruby to get an understanding and impression what reflection in various programming language and runtime environment means.

2. Based on that experience we have defined the meaning of reflection and a corresponding library. Additionally, we have specified a minimal set of reflection facilities that a reflection library should provide.

3. Based on these definitions and specifications, we have designed a reflection library for Eiffel (ERL).

4. After that, we have implemented ERL partly. The facilities provided in INTERNAL restricted the implementation.

5. At last, we have built the ObjectBrowser to traverse object structures to test ERL.

We achieved the intended results.

5.2 Main contribution

Existing reflection APIs

Beside Smalltalk, none of the evaluated reflection libraries map the object model of the corresponding programming language or runtime environment correctly. Mapping the object model is almost independent from the functional range of a reflection library, nevertheless it should be a part of reflection. It would be more convenient if we were able to use a reflection API by following the object model.

Smalltalk does not only provide the largest set of functionality, but also a strict separation of object and class features. The main reason is that Smalltalk and its runtime system is written in itself and therefore needs reflection to work.
Eiffel reflection library

ERL maps the Eiffel object model and provides an interface with which we can retrieve common known reflection information of objects. In other words, ERL fulfills the definitions and specification made in section 3.1.

The implementation of ERL is restricted to the facilities provided in INTERNAL. Therefore, it is not possible to get information about routines, deferred features, parent types, and formal generic parameters of generic classes. Some other problems with INTERNAL occurred while developing the library. These situations are described in section 3.4 more precisely.

ObjectBrowser

As a use case for ERL, we have build the ObjectBrowser. We are able to traverse given object structures and modify their basic type attributes. Additionally, the ObjectBrowser is able to lay out the object structure in a graph.

A user guide describes how to use the ObjectBrowser.

5.3 Future work

To support all facilities of ERL, the following problems must be considered in a forthcoming Eiffel release.

1. The problems encountered with INTERNAL must be addressed and a satisfiable solution must be implemented.

2. The missed facilities of the Eiffel environment to support a complete reflection library must be implemented.

3. It would make sense to integrate ERL into the Eiffel base library and maybe adapt ANY to get at least the type of an object.

4. When the first two point above are fulfilled, the ObjectBrowser must be adapted to let it show all possible information from ERL.
Appendix A

ERL API

A.1 ERL_ARGUMENT

indexing

  description: "Representation of a formal argument of a routine to reflect its properties."
  revision: "$Revision: 1.10 $"
  author: "Beat Fluri"
  license: "Eiffel Forum License V2 (see license.txt)"

defered class ERL_ARGUMENT

feature -- Access

  name: STRING
    -- Name of this argument

  position: INTEGER
    -- Position of this argument in the argument list of its routine

  routine: ERL_ROUTINE
    -- Routine to which this argument belongs

  type: ERL_TYPE
    -- Type of this argument

invariant

  name_not_empty: name /= Void and then not name.is_empty
  routine_exists: routine /= Void
  valid_position: position > 0

end -- class ERL_ARGUMENT
A.2  **ERL_ATTRIBUTE**

indexing

*description:* "Representation of an attribute to reflect its properties."
*revision:* "$Revision: 1.10 "$
*author:* "Beat Fluri"
*license:* "Eiffel Forum License V2 (see license.txt)"

**deferred class ERL_ATTRIBUTE**

**inherit**

  **ERL_QUERY**

  **rename**

  *result_type:* as *attribute_type*

 **end**

**feature** -- Status report

  *Is_deferred:* BOOLEAN is **False**

  -- Attributes are always effective

**feature** -- Element change

  **replace** (*a_value:* ANY)

  -- Replace value of this attribute to 'a_value'.

  **require**

  *object_exists:* is *object_set*

  *a_value_exists:* *a_value /= Void*

  *a_value_type_conforms_attribute_type:* erl_result_type_conforming_to (*a_value*)

 **end**  -- class ERL_ATTRIBUTE
A.3  **ERL_CLASS**

**indexing**

*description:* "Representation of a class to reflect its properties."

*revision:* "$Revision: 1.10 $"

*author:* "Beat Fluri"

*license:* "Eiffel Forum License V2 (see license.txt)"

**deferred class** **ERL_CLASS**

**feature** -- Access

- **cluster**: **ERL_CLUSTER**
  -- Cluster which contains this type

- **formal generics**: **LIST [ERL GENERIC]**
  -- Formal generic parameter of this class

- **name**: **STRING**
  -- Name of this class

**feature** -- Status report

- **conform generic constraints** ([a_actuals]: **LIST [ERL_TYPE]**): **BOOLEAN**
  -- Does actuals ‘a_actuals’ conform to generic constraints of this class?

- **exists type with actuals** ([a_actuals]: **LIST [ERL_TYPE]**): **BOOLEAN**
  -- Does type derived from this class with actual generics ‘a_actuals’ exist?

- **is_generic**: **BOOLEAN**
  -- Is this class a generic class?

**feature** -- Basic operation

- **type with actuals** ([a_actuals]: **LIST [ERL_TYPE]**): **ERL_TYPE**
  -- Type derived from this class with actual generic parameters ‘a_actuals’.

  **require**
  - **is_class_generic**: is_generic
  - **a_actuals conform constraints**: conform generic constraints ([a_actuals])

**invariant**

- **name not empty**: name /= Void and then not name.is_empty

- **formal generics**: formal generics /= Void and
  - **is_generic implies not formal generics is empty**

- **cluster exists**: cluster /= Void

**end** -- class **ERL_CLASS**
Chapter A. ERL API

A.4 ERL_CLUSTER

indexing
  description: "Representation of a cluster to reflect its properties."
  revision: "$Revision: 1.10 $"
  author: "Beat Fluri"
  license: "Eiffel Forum License V2 (see license.txt)"

defered class ERL_CLUSTER

feature -- Access

  classes: SUBSET [ERL_CLASS]
    -- All classes in this cluster

  name: STRING
    -- Absolut name of this cluster

  parent: ERL_CLUSTER
    -- Parent of this cluster

feature -- Status report

  is_root_cluster: BOOLEAN
    -- Is this cluster a root cluster?

invariant
  name_not_empty: name /= Void and then not name.is_empty
  not_root_cluster: not is_root_cluster implies parent /= Void
  root_cluster: is_root_cluster implies parent = Void

end -- class ERL_CLUSTER
A.5  

**ERL\_CONSTRAINT**

indexing

*description:* "Representation of a generic constraint to reflect its properties."

*revision:* "\$Revision: 1.10 \$

*author:* "Beat Fluri"

*license:* "Eiffel Forum License V2 (see license.txt)"

**deferred class** ERL\_CONSTRAINT

**feature** -- Access

\[\text{class\_bound: ERL\_CLASS} \]

\[\text{-- Class to which this constraint is bound}\]

\[\text{creators: SUBSET [ERL\_ROUTINE]} \]

\[\text{-- Creators allowed for this constraint}\]

\[\text{type\_bound: ERL\_TYPE} \]

\[\text{-- Type to which this constraint is bound}\]

**feature** -- Status report

\[\text{has\_creator\_constraints: BOOLEAN} \]

\[\text{-- Has this constraint restricted creators?}\]

\[\text{is\_bound\_type: BOOLEAN} \]

\[\text{-- Is bound a type?}\]

**invariant**

\[\text{creators\_exists: creators /= Void} \]
\[\text{creator\_constraints: has\_creator\_constraints implies not creators.is\_empty} \]
\[\text{class\_bound\_exists: class\_bound /= Void} \]
\[\text{type\_bound: (is\_bound\_type implies type\_bound /= Void) and then} \]
\[\text{class\_bound = type\_bound.base\_class} \]

**end** -- class ERL\_CONSTRAINT
A.6 **ERL_FEATURE**

**indexing**
- **description**: "Representation of a feature (routine, attribute) to reflect its properties."
- **revision**: "$Revision: 1.10 $"
- **author**: "Beat Fluri"
- **license**: "Eiffel Forum License V2 (see license.txt)"

**deferred class** **ERL_FEATURE**

**feature** -- Access

- **category**: STRING
  -- Header comment of this feature
  **ensure**
  - **result_set**: Result = feature_clause.category

- **clients**: SUBSET [ERL_CLASS]
  -- Feature availability (export status)
  **ensure**
  - **result_set**: Result = feature_clause.clients

- **effective_parent**: ERL_FEATURE
  -- Effective feature in one of the parent classes if inherited and not redefined

- **feature_clause**: ERL_FEATURE_CLAUSE
  -- Feature clause of this feature

- **names**: SUBSET [STRING]
  -- Names of this feature

- **parents**: BAG [ERL_FEATURE]
  -- Features in parent class if inherited

**postcondition**: BOOLEAN

- -- Are postconditions satisfied invoking this feature?
  **require**
  - postcondition.check_available: can_postcondition_be_checked

**precondition** (a_operands: TUPLE): BOOLEAN

- -- Do 'a_operands' satisfy features's precondition?
  **require**
  - a_operands.exist: a_operands /= Void
  - a_operands_valid: valid_operands (a_operands)
feature -- Status report

  can_postcondition_be_checked: BOOLEAN
  -- Can postcondition be checked?

  erl_type_conforming_to (other: ANY): BOOLEAN
  -- Conforms ‘other’ to the type this feature belongs?
  require
    other_exists: other /= Void

  is_deferred: BOOLEAN
  -- Is this feature deferred?

  is_frozen: BOOLEAN
  -- Is this feature frozen?

  is_inherited: BOOLEAN
  -- Is this feature inherited?

  is_object_set: BOOLEAN
  -- Is ‘object’ set?
  ensure
    result_set: Result = (object /= Void)

  isObsolete: BOOLEAN
  -- Is this feature obsolete?

  is_redefined: BOOLEAN
  -- Is this feature redefined?

  is_renamed: BOOLEAN
  -- Is this feature renamed?

  is_selected: BOOLEAN
  -- Is this feature selected?

  names_not_empty (a_names: SUBSET [STRING]): BOOLEAN
  -- Are all names in ‘a_names’ not empty?
  require
    a_names_exist: a_names /= Void

  valid_operands (a_operands: TUPLE): BOOLEAN
  -- Are ‘a_operands’ valid operands?
  require
    a_operands_exist: a_operands /= Void
feature -- Status setting

set_object (a_object: ANY)
   -- Set 'object' to 'a_object'.
   require
      a_object_exists: a_object /= Void
      a_object_conforms: erl_type_conforming_to (a_object)
   ensure
      object_set: object = a_object

invariant
   names_set: names /= Void and then not names.is_empty and then
      names_not_empty (names)
   feature_clause_exists: feature_clause /= Void
   parents_exists: parents /= Void
   inheritance: ((is_renamed or is_redefined or is_selected) implies is_inherited)
      and
      (is_inherited implies not parents.is_empty)
      and
      (is_redefined implies effective_parent = Void)

end -- class ERL_FEATURE
A.7 **ERL\_FEATURE\_CLAUSE**

**indexing**
- **description**: "Representation of a feature clause to reflect its properties."
- **revision**: "$Revision: 1.10 "$
- **author**: "Beat Fluri"
- **license**: "Eiffel Forum License V2 (see license.txt)"

**deferred class** **ERL\_FEATURE\_CLAUSE**

**feature** -- Access

  - **category**: STRING
    -- Header comment of this feature clause
  
  - **clients**: SUBSET [ERL\_CLASS]
    -- Feature availability (export status)
  
  - **features**: SUBSET [ERL\_FEATURE]
    -- All features of this feature clause.
    -- Empty set, not Void, if this feature clause has no features
  
  **ensure**
  
  - **result\_set**: Result /= Void

  **type**: ERL\_TYPE
  -- Type to which this feature clause belongs

**feature** -- Status report

  - **has\_clients**: BOOLEAN
    -- Is this feature clause exported to special clients?

**invariant**

  - **category\_not\_empty**: category /= Void implies not category.is\_empty
  
  - **type\_exists**: type /= Void

  - **clients\_exists**: clients /= Void

  - **no\_export\_status**: not has\_clients implies clients.is\_empty

  - **export\_status**: has\_clients implies not clients.is\_empty

**end** -- class **ERL\_FEATURE\_CLAUSE**
A.8  \texttt{ERL\_FUNCTION}

\textbf{indexing}
\begin{itemize}
\item \textit{description}: "Representation of a function to reflect its properties."
\item \textit{revision}: "$\$Revision: 1.10 \$
\item \textit{author}: "Beat Fluri"
\item \textit{license}: "Eiffel Forum License V2 (see license.txt)"
\end{itemize}

\textbf{deferred class} \texttt{ERL\_FUNCTION}

\textbf{inherit}
\begin{itemize}
\item \texttt{ERL\_ROUTINE}
\item \texttt{ERL\_QUERY}
\end{itemize}

\textbf{feature} -- Access

\begin{itemize}
\item \texttt{effective\_parent}: \texttt{ERL\_FUNCTION}
\item \texttt{item}: \texttt{ANY}
\item \texttt{parents}: \texttt{BAG [ERL\_FUNCTION]}
\end{itemize}
A.9  **ERL\_GENERIC**

indexing

*description*: "Representation of a formal generic parameter to reflect its properties."

*revision*: "$Revision: 1.10 "$

*author*: "Beat Fluri"

*license*: "Eiffel Forum License V2 (see license.txt)"

**deferred class**  **ERL\_GENERIC**

**feature**  --  Access

*constraints*: LIST[ERL\_CONSTRAINT]
  -- Generic constraints of this generic parameter

*name*: STRING
  -- Formal generic parameter

**feature**  --  Status report

*has\_constraints*: BOOLEAN
  -- Has this generic parameter constraints?

**invariant**

*name\_not\_empty*: name / Void and then not name.is.empty

*constraints\_exist*: constraints /= Void

*generic\_constraints*: has\_constraints implies not constraints.is.empty

**end**  --  class  **ERL\_GENERIC**
indexing
description: "Representation of a procedure to reflect its properties."
revision: "$Revision: 1.10 $"
author: "Beat Fluri"
license: "Eiffel Forum License V2 (see license.txt)"

defered class ERLPROCEDURE

inherit
ERLROUTINE

rename
routine as procedure

redefine
parents,
effective_parent,
procedure

feature -- Access

effective_parent: ERLPROCEDURE
-- See '{ERLFEATURE}.effective_parent' for more details.

parents: BAG [ERLPROCEDURE]
-- See '{ERLFEATURE}.parents' for more details.

feature -- Status report

is_creator: BOOLEAN
-- Is this procedure a creator?

end -- class ERLPROCEDURE
A.11  **ERL_QUERY**

**indexing**

*description*: "Representation of a query (function or attribute) to reflect its properties."

*revision*: "$Revision: 1.10 $"

*author*: "Beat Fluri"

*license*: "Eiffel Forum License V2 (see license.txt)"

**deferred class** **ERL_QUERY**

**inherit**

**ERL_FEATURE**

**redefine**

parents,

*effective_parent*

**end**

**feature**  --  Access

*effective_parent*: **ERL_QUERY**

  -- See ‘{**ERL_FEATURE**}.effective_parent’ for more details.

*item*: **ANY**

  -- Value of this attribute

**require**

*object_exists*: *is_object_set*

**ensure**

*result_type_conforms*: *result_type*: \(\text{Result} \neq \text{Void}\) implies

*erl_result_type_conforming_to* (Result)

*parents*: **BAG** [**ERL_QUERY**]

  -- See ‘{**ERL_FEATURE**}.parents’ for more details.

*result_type*: **ERL_TYPE**

  -- Type of this attribute

**feature**  --  Status report

*erl_result_type_conforming_to* (other: **ANY**): **BOOLEAN**

  -- Conforms ‘other’ to the result type?

**require**

*other_exists*: other \(\neq\) Void

**end**  --  class **ERL_QUERY**
A.12 **ERL_ROUTINE**

indexing

description: "Representation of a routine to reflect its properties."
revision: "$Revision: 1.10 $"
author: "Beat Fluri"
license: "Eiffel Forum License V2 (see license.txt)"

defered class **ERL_ROUTINE**

inherit

**ERL_FEATURE**

rename

can_postcondition_be_checked as was_call_made

redefine

parents,
effective_parent,
was_call_made

end

feature -- Access

effective_parent: **ERL_ROUTINE**

-- See `{ERL_FEATURE}.effective_parent` for more details.

formal_arguments: LIST [ERL_ARGUMENT]

-- All formal arguments of this routine.

operands: TUPLE

-- Actual operands with which this routine operates

parents: BAG [ERL_ROUTINE]

-- See `{ERL_FEATURE}.parents` for more details.

postcondition: BOOLEAN

-- Are postconditions satisfied after a call was made?

precondition (a_operands: TUPLE): BOOLEAN

-- Do `a_operands` satisfy routine's precondition?

feature -- Status report

is_deferred: BOOLEAN

-- Is this routine deferred?
\textit{valid\_operands} (a\_operands: TUPLE): BOOLEAN
\hspace{1em} -- Are 'a\_operands' valid operands, i.e. conform they to the 'formal\_arguments'?

\textit{was\_call\_made}: BOOLEAN
\hspace{1em} -- Was a call made?

\textbf{feature} -- Element change

\textit{set\_operands} (args: TUPLE)
\hspace{1em} -- Set 'operands' to 'args'.
\textbf{require}
\hspace{1em} args\_exist: args /= Void
\hspace{1em} valid\_args: valid\_operands (args)
\textbf{ensure}
\hspace{1em} operands\_set: (operands /= Void implies equal (operands, args)) or
\hspace{1em} (operands = Void implies (args = Void or else args\_is\_empty))

\textbf{feature} -- Basic operations

\textit{call}
\hspace{1em} -- Call routine.
\textbf{require}
\hspace{1em} not\_is\_deferred: not is\_deferred
\hspace{1em} operands\_set: operands /= Void
\textbf{ensure}
\hspace{1em} a\_call\_was\_made: was\_call\_made = True

\textbf{invariant}
\hspace{1em} set\_operands\_are\_valid: operands /= Void implies valid\_operands (operands)
\hspace{1em} routine\_exists: routine /= Void

\textbf{end} -- class \textit{ERL\_ROUTINE}
A.13  **ERL_TYPE**

indexing
  
  description: "Representation of a type to reflect its properties."
  
  revision: "$Revision: 1.10 $"
  
  author: "Beat Fluri"
  
  license: "Eiffel Forum License V2 (see license.txt)"

**deferred class** **ERL_TYPE**

**feature** -- Access

  **actual generics**: LIST [ERL_TYPE]
  
  -- Actual generic parameters of this type

  **attributes**: SUBSET [ERL_ATTRIBUTE]
  
  -- All attributes of this type

  **ensure**
  
  result_exists: Result /= Void

  **base class**: **ERL_CLASS**
  
  -- This type's base class

  **client context**: **ERL_CLASS**
  
  -- Client context from which this type is looked at

  **deferred features**: SUBSET [ERL_FEATURE]
  
  -- All deferred features of this type

  **ensure**
  
  result_exists: Result /= Void

  **feature by name** (**a name**: STRING): **ERL_FEATURE**
  
  -- Search feature with name ‘a name’.

  **feature clauses**: SUBSET [ERL_FEATURE_CLAUSE]
  
  -- All feature clauses of this type

  -- Empty set, not Void, if this type has no feature clauses

  **ensure**
  
  result_set: Result /= Void

  **features**: SUBSET [ERL_FEATURE]
  
  -- All features of this type

  -- Empty set, not Void, if this type has no features

  **ensure**
  
  result_set: Result /= Void

**invariants** (**an object**: ANY): **BOOLEAN**
A.13  ERL_TYPE

-- Does 'an_object' fulfill the invariants of this type?

require
  an_object_exists: an_object /= Void

lookup_binding (a_feature: ERL_FEATURE): ERL_FEATURE
  -- Feature to call according to vtable, when calling 'a_feature' in parent type.

require
  a_feature_exists: a_feature /= Void

name: STRING
  -- Name of this type without any generic information

parents: BAG [ERL_TYPE]
  -- All direct parent types of this type

routines: SUBSET [ERL_ROUTINE]
  -- All routines of this type

ensure
  result_exists: Result /= Void

feature -- Status report

erl_conforms_to (other: ERL_TYPE): BOOLEAN
  -- Does this erl type conforms to 'other'?

require
  other_exists: other /= Void

has_actual_generics: BOOLEAN
  -- Has this type actual generic parameters?

is_basic_type: BOOLEAN
  -- Is this type a basic type?

is_deferred: BOOLEAN
  -- Is this type deferred?

is_expanded: BOOLEAN
  -- Is this type expanded?

is_obsolete: BOOLEAN
  -- Is this type obsolete?
feature  --  Status setting

    set_client_context (a_client_context: ERL_CLASS)
    --  Set ‘client_context’ to ‘a_client_context’.
    require
        a_client_context_exists: a_client_context /= Void
    ensure
        client_context_set: client_context = a_client_context

feature  --  Basic operation

    new_instance: ANY
    --  New instance of this type.
    require
        type_not_deferred: not is_deferred
    ensure
        result_set: Result /= Void

invariant

    name_not_empty: name /= Void and then not name.is_empty
    base_class_exists: base_class /= Void
    parents_not_empty: parents /= Void and then
        (not name.is_equal ("ANY") implies not parents.is_empty)
    actual.generics_exist: actual.generics /= Void
    actual.generics_not_empty: base_class.is_generic implies not actual.generics.is_empty
    feature_clauses_exist: feature_clauses /= Void
    features_exist: features /= Void
    feature_clauses_feature_clauses_relation: feature_clauses.is_empty implies features.is_empty
    default_client_context: client_context /= Void

end  --  class ERL_TYPE
A.14  ERL_UNIVERSE

indexing

description: "Objects that give access to ERL"
revision: "$Revision: 1.10 $"
author: "Beat Fluri"
license: "Eiffel Forum License V2 (see license.txt)"

class ERL_UNIVERSE

feature -- Access

cluster_by_name (a_name: STRING): ERL_CLUSTER
   -- Search cluster by name 'a_name'.

custers: SUBSET [ERL_CLUSTER]
   -- All clusters in the system

objects: SUBSET [ANY]
   -- All objects in the system

type_by_name (a_name: STRING): ERL_TYPE
   -- Search type by name 'a_name'.

types: SUBSET [ERL_TYPE]
   -- All used types in the system

feature -- Status report

exists_type_for_name (a_name: STRING): BOOLEAN
   -- Does type for name 'a_name' exist?
require
   a_name_not_empty: a_name /= Void and then not a_name.is_empty

feature -- Basic operation

class_for_name (a_name: STRING): ERL_CLASS
   -- Clss for name 'a_name'
require
   a_name_not_empty: a_name /= Void and then not a_name.is_empty

object_type (a_object: ANY): ERL_TYPE
   -- Type of object 'a_object'
require
   a_object_exists: a_object /= Void
ensure
   result_exists: Result /= Void
type for name (a name: STRING): ERL_TYPE
   -- Type for name 'a_name'
   require
      a_name_not_empty: a_name /= Void and then not a_name.is_empty
      type_exists: exists_type_for_name (a_name)
   ensure
      result_exists: Result /= Void
end -- class ERL_UNIVERSE
Appendix B

Scope of the ERL implementation

The following table shows an overview of the implemented, partially implemented and not implemented reflection facilities in ERL.

<table>
<thead>
<tr>
<th>Class</th>
<th>Feature</th>
<th>impl.</th>
<th>part.</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERL_UNIVERSE</td>
<td><code>class_for_name</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>cluster_by_name</code></td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>clusters</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>exists_type_for_name</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>object_type</code></td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>objects</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>type_by_name</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>type_for_name</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>types</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>ERL_TYPE</td>
<td><code>actual_generics</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>attributes</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>base_class</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>client_context</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>deferred_features</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>erl_conforms_to</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>features</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>feature_by_name</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>feature_clauses</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>has_actual_generics</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>invariants</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>is_basic_type</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>is_deferred</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>is_expanded</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>is_obsolete</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lookup_binding</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>name</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>new_instance</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>parents</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>routines</code></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td><code>set_client_context</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Chapter B. Scope of the ERL implementation

<table>
<thead>
<tr>
<th>Interface</th>
<th>Feature</th>
<th>impl.</th>
<th>part.</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERL_CLUSTER</td>
<td>classes</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_root_cluster</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>ERL_CLASS</td>
<td>cluster</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exists_type_with_actuals</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>formal_generic_parameters</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>generic_constraints</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_generic</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type_with_actuals</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL_GENERIC</td>
<td>constraints</td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>has_constraints</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERLCONSTRAINT</td>
<td>class_bound</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creators</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has_creator_constraints</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_bound_type</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type_bound</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL_FEATURE_CLAUSE</td>
<td>category</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clients</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>features</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has_clients</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>ERL_FEATURE</td>
<td>can_postcondition_be_checked</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>category</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clients</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>effective_parent</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>erl_type_conforming_to</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>feature_clause</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_deferred</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_frozen</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_inherited</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isObsolete</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_object_set</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_redefined</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_renamed</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_selected</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>names</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parents</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>postconditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>preconditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_object</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>valid_operands</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>Feature</td>
<td>impl.</td>
<td>part.</td>
<td>not</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>ERL_ROUTINE</td>
<td>call</td>
<td>×</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>formal_arguments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>operands</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_operands</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>valid_operands</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL_ARGUMENT</td>
<td>name</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>position</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>routine</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL_PROCEDURE</td>
<td>is_creator</td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>ERL_FUNCTION</td>
<td>item</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result_type</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL_ATTRIBUTE</td>
<td>attribute_type</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>item</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>replace</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 additional to feature
2 additional to routine
References


