Chair of Software Engineering

JEiffel – A Java Bytecode Generator Backend for the ISE Eiffel Compiler

Diploma Thesis

By: Benno Baumgartner

Supervised by: Till G. Bay
Prof. Bertrand Meyer

Student Number: 98-727-589

ETH

inf | Informatik
Computer Science

Abstract

Eiffel Software[a] provides a compiler for the Eiffel programming language capable of generating C code or Common Intermediate Language byte code. The CLI code can be executed on the Common Language Runtime, the core of Microsofts .Net technology. The goal of the diploma thesis is to map Eiffel to Java and to write a Java Byte Code generator backend for the ISE compiler to generate Java Byte Code for a subset of Eiffel. The generated code can be executed on a Java Virtual Machine.
A short overview is given in chapter 1. The high level mapping is presented in chapter 2. The implementation of the Java Byte Code generator in chapter 3.
The language constructs of Eiffel which are not yet implemented are shown in chapter 4. An example application along with parts of the generated Java Byte Code is shown in chapter 5. The result of this proof of concept is shown in chapter 6.

[a] `http://www.eiffel.com`

Contents

1 Introduction **1**

1.1 Goal . . . 1

1.2 Overview . . . 1

1.3 Example . . . 1

1.4 Implementation . . . 4

1.5 How to read this report . . . 4

2 Concept **5**

2.1 Overview . . . 5

2.2 Mapping classes . . . 5

2.2.1 Mapping a single class . . . 5

2.2.2 Single inheritance . . . 6

2.2.3 Multiple inheritance . . . 7

2.2.4 Renaming . . . 8

2.2.5 Name mangling . . . 16

2.2.6 Constructors . . . 17

2.2.7 Sharing . . . 18

2.2.8 Join with select . . . 19

2.2.9 Attributes . . . 21

2.2.10 Sharing of attributes . . . 23

2.2.11 Inheriting multiple times from the same class . . . 25

2.2.12 Constants . . . 27

2.2.13 Redefining and Precursors . . . 27

2.2.14 Deferred features . . . 28

2.2.15 Undefine . . . 29

2.2.16 Covariance . . . 29

2.2.17 Export statement . . . 31

2.2.18 Friends . . . 33

2.2.19 Expanded Types . . . 35

2.2.20 once . . . 36

2.3 Basic types . . . 37

2.4 Code . . . 39

2.5 Contracts . . . 39

2.5.1 Checks . . . 39

2.5.2 Preconditions . . . 39

2.5.3 Postcondition . . . 40

2.5.4 Loop variants and invariants . . . 41
2.5.5 Class invariants . . . 43

3 Implementation **45**
3.1 Overview . . . 45
3.2 Mapper . . . 45
3.3 Name mangling . . . 46
3.4 Generation . . . 48
3.4.1 Generating interfaces . . . 49
3.4.2 Generating classes . . . 51
3.4.3 Generating code . . . 52

4 Missing parts **55**

5 Case study **57**
5.1 Overview . . . 57
5.2 Design . . . 57
5.3 Implementation . . . 58
5.3.1 Eiffel side . . . 58
5.3.2 Java side . . . 62
5.4 Translation . . . 64

6 Conclusion **67**
6.1 Overview . . . 67
6.2 From Eiffel to Java . . . 67
6.3 Implementation . . . 68
6.4 Outlook . . . 68

Chapter 1

Introduction

1.1 Goal

The goal of the project is to compile Eiffel programs to Java Byte Code with the constrain that the generated code must run on a standard Java Virtual Machine[a]. Java programmers should be able to use the generated classes in their Java programs. In particular it should be possible to inherit from the generated classes or use them as a client. The generated code should not loose the information about the subtyping structure of the Eiffel classes. Therefore the goal is not only to compile to Java Byte Code and use the JVM as a machine but also to map Eiffel as closely as possible to Java to allow a tight integration of the two languages. This is different from the approach chosen by the SmartEiffel Team in the sense that their Java Byte Code generator does not map type relations to Java but flatten each Eiffel class and generates one Java class out of this [2].

1.2 Overview

The main challenge in mapping Eiffel to Java is multiple inheritance which is an integral part of Eiffel. Multiple inheritance is not supported in Java, but multiple inheritance can be modelled in Java with the help of interfaces. Multiple subclassing can be implemented through either code duplication or delegation.
Multiple subtyping means that a type can have more then one direct super type. Multiple subclassing means that a class can inherit code from multiple other classes directly. In Eiffel as defined in Eiffel: the language [5] subclassing and subtyping is not separated from each other. Inheriting from another class is a subclassing and a subtyping. Some Eiffel dialects allow to import another class, this is a subclassing but not a subtyping. Extending or implementing a Java interface is a subtyping but not a subclassing.

1.3 Example

For every Eiffel class a Java interface is generated with the same set of members as the Eiffel class. The interface reflects the subtyping relation of the Eiffel class. For every interface a Java class implementing the interface is generated. This class contains the generated code from the

[a]Version 1.4.2 or higher.

corresponding Eiffel class.
The class *COWBOY_PICTURE* in the following example inherits from *COWBOY* and from class *PICTURE* (σ_0 and σ_1 denotes a sequence of statements).

```
class COWBOY
feature
        shoot is do σ0 end
end

class PICTURE
feature
        draw is do σ1 end
end

class COWBOY_PICTURE
inherit
        COWBOY
        PICTURE
end

        local
                cp: COWBOY_PICTURE
        do
                create cp
                cp.shoot -- yields σ0
                cp.draw -- yields σ1
        end
```

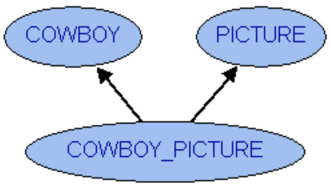

Figure 1.1: Multiple inheritance in Eiffel.

The following listing shows how the Eiffel classes can be translated to Java.

```
public interface Cowboy extends Any {
    public void shoot();
}

public interface Picture extends Any {
    public void draw();
}

public interface CowboyPicture extends Cowboy, Picture {
}

public class CowboyImpl implements Cowboy {
    public void shoot() {σ0}
}

public class PictureImpl implements Picture {
    public void draw() {σ1}
}

public class CowboyPicuteImpl implements CowboyPicture {

    private CowboyImpl cowboyDelegate = new CowboyImpl();
    private PictureImpl pictureDelegate = new PictureImpl();

    public void draw() {
        pictureDelegate.draw();
    }

    public void shoot() {
        cowboyDelegate.shoot();
    }
}

    CowboyPicture cp = new CowboyPictureImpl();
    cp.shoot(); // yields σ0
    cp.draw(); // yields σ1
```

The object referenced by cp is of type: **CowboyPictureImpl**, **CowboyPicture**, **Cowboy**, **Picture**, **Any**, and **Object** and a call to shoot() respectively draw() will execute the code sequences in **CowboyImpl** respectively **PictureImpl**. Every generated interface is a subtype of **Any** and **Any** is a subtype of **Object**.

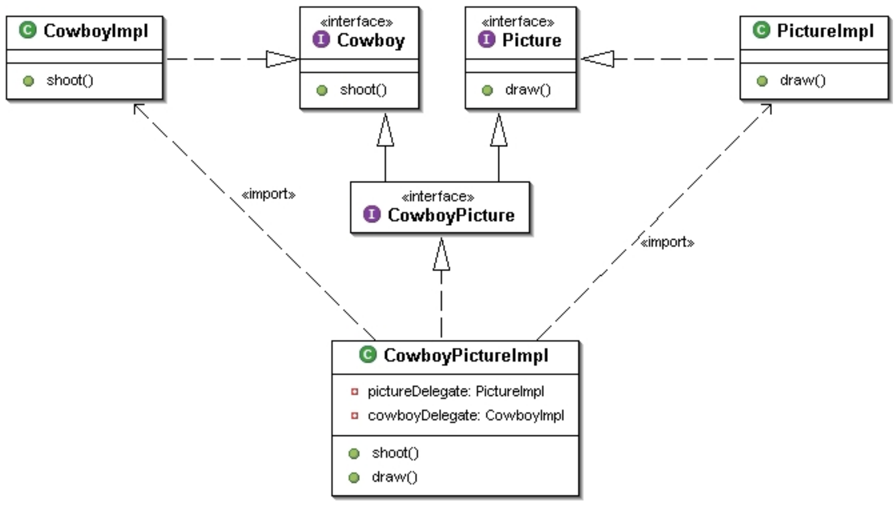

Figure 1.2: Multiple inheritance in Java.

1.4 Implementation

The translation of the Java program above to Java Byte Code is straight forward. In most parts of the report Java source code is shown and not Java Byte Code although the compiler backend does not generate Java source code. But Java source code is much easier to read and if it is possible to translate an Eiffel language construct to Java source code then it can also be translated to Java Byte Code.
The ISE compiler builds an AST for every Eiffel class if the class is valid or produces a compiler error message otherwise. The Java Byte Code generator uses the same visitor on the AST as the CIL generator. There are two main steps performed by the Java Byte Code generator:

1. Generate the Java interfaces and the Java class skeletons from the Eiffel class.

2. Translate the non deferred features of an Eiffel class to Java Byte Code.

The Java Byte Code generator is written in Eiffel. It makes extensive use of the Eiffel Library to generate Java Byte Code [4].

1.5 How to read this report

This report has to deal with two different languages at the same time: Eiffel and Java. Although both languages are object oriented and share a lot of properties there are differences. To minimise the ambiguities the differences in terminology are listed below. The report tries to use Eiffel terminology whenever it talks about Eiffel and Java terminology whenever it talks about Java. Below is a table that compares the Eiffel terminology with the terminology used in Java. It also shows a short description of each term. It can not give a deep introduction into the semantics behind each term. For Eiffel these terms are all discussed in depth in Object-Oriented Software Construction [6] or Eiffel: the language [5]. For Java a discussion can be found in Java in a nutshell [3].

Eiffel	Java	Description
class	class	A class is an implementation of an abstract data type.
object	object	An object is an instance of a class. It exists only at runtime.
-	interface	An interface can be understood as a completely abstract class. There is no such thing as an interface in Eiffel.
deferred	abstract	A deferred feature is a feature without an implementation. A class containing a deferred feature has to be deferred.
feature	member	Attributes and routines of a class.
routine	method	A feature with a body. It does calculate something.
attribute	field	A feature without a body. It holds a value.
query	function or field	A feature with a return value.
command	procedure	A feature without a return value.
creation feature	constructor	A feature which can be used to instantiate an object.
class interface	-	A view of a class which shows the class without its code.

To increase the readability all Eiffel classes do have a different fonts and colors than Java classes throughout the text:
AN_EIFFEL_CLASS **AJavaClass**
All Eiffel features are written in italic whereas Java methods are not:
an_eiffel_feature aJavaMethod()

Chapter 2

Concept

2.1 Overview

This chapter presents the mapping from Eiffel to Java. The section 2.2 is about features and how to dispatch feature calls to the right implementation in Java. It shows how renaming, redefining, exporting, sharing, covariance and other Eiffel language constructs are handled. The mapping for each of this constructs is explained in a subsection.
Section 2.3 shows how the so called Eiffel basic types are mapped to the Java primitive types.
Section 2.4 shows how to translate code of feature bodies to Java Byte Code.
Section 2.5 is about how contracts are translated to Java.

2.2 Mapping classes

This section shows how to map an Eiffel class to Java. This is challenging because Java does not support multiple inheritance and even worse there is no way of declaring a public method as non virtual[a]. A virtual method is a polymorphic one.

2.2.1 Mapping a single class

For every Eiffel class a Java interface is generated. The interface contains the same members as the Eiffel class. A Java class implementing this interface and containing the translated code from the Eiffel class is then generated. The implementation must yield the same result as the implementation in the Eiffel class. The simple Eiffel class:

```
class PICTURE
feature
        draw is do σ0 end
end
```

Is translated to the following Java **interface** and **class**.

```
public interface Picture extends Any {
    public void draw();
}

```

[a]Unless it is declared as static

```
public class PictureImpl implements Picture {
    public void draw() {σ0}
}
```

It is important to define the types of variables in Java only by using the Interface since only the interfaces have subtyping relations among each other:

```
    Picture picture = new PictureImpl();
    picture.draw();
```

2.2.2 Single inheritance

In Java only the interfaces will have subtyping relations: These will match those from Eiffel exactly. The implementations of the **interfaces** do not have any subtyping relations. This approach allows to model multiple inheritance later. Let us take as an example an additional Eiffel class *PNG_PICTURE* which inherits from the *PICTURE* as described in the subsection 2.2.1:

```
class PNG_PICTURE
inherit
        PICTURE
feature
        compress is do σ1 end
end
```

Let us translate that again to Java:

```
public interface PngPicture extends Picture {
    public void compress();
}

public class PngPictureImpl implements PngPicture {

    private PictureImpl pictureDelegate = new PictureImpl();

    public void compress() {σ1}
    public void draw() {pictureDelegate.draw();}
}
```

The Java class **PngPictureImpl** is of type **PngPictureImpl**, **PngPicture**, **Picture**, **Any** and **Object** but not of type **PictureImpl**. Since *PNG_PICTURE* did not redefine *draw* from *PICTURE*, **PngPictureImpl** can delegate a call to draw() to the implementation of draw() in **PictureImpl**.
Below is an example how to use these classes:

```
1       Picture picture = new PictureImpl();
2       PngPicture pngPicture = new PngPictureImpl();
3       picture.draw();          //yields σ0
4       pngPicture.draw();       //yields σ0
5       pngPicture.compress();   //yields σ1
6       picture = pngPicture;
7       picture.draw();          //yields σ0
```

The assignment at line 6 is possible because the object referenced by pngPicture is a subtype of **Picture**.

2.2.3 Multiple inheritance

Even though Java does not support multiple inheritance simulating it is possible because Java does support multiple subtyping through interfaces natively and subclassing can always be replaced by even code duplication or delegation. Let us take the Eiffel classes:

```
class PICTURE
feature
        draw is do σ0 end
end

class COWBOY
feature
        shoot is do σ1 end
end

class COWBOY_PICTURE
inherit
        PICTURE
        COWBOY
end

        local
                cowboy_picture: COWBOY_PICTURE
        do
                create cowboy_picture
                cowboy_picture.draw  -- yields σ0
                cowboy_picture.shoot -- yields σ1
        end
```

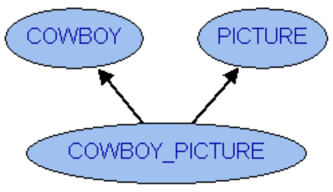

Figure 2.1: Multiple inheritance in Eiffel.

The translation to Java using the pattern introduced in the previous section 2.2.2 is shown next.

```
public interface Picture extends Any {
    public void draw();
}

public interface Cowboy extends Any {
    public void shoot();
}

public interface CowboyPicture extends Picture, Cowboy {}


public class PictureImpl implements Picture {
    public void draw() {$\sigma_0$}
}

public class CowboyImpl implements Cowboy {
    public void shoot() {$\sigma_1$}
}

public class CowboyPictureImpl implements CowboyPicture {

    private PictureImpl pictureDelegate = new PictureImpl();
    private CowboyImpl cowboyDelegate = new CowboyImpl();

    public void shoot() {cowboyDelegate.shoot();}
    public void draw() {pictureDelegate.draw();}
}

    CowboyPicture cowboyPicture = new CowboyPictureImpl();
    cowboyPicture.draw();
    cowboyPicture.shoot();
```

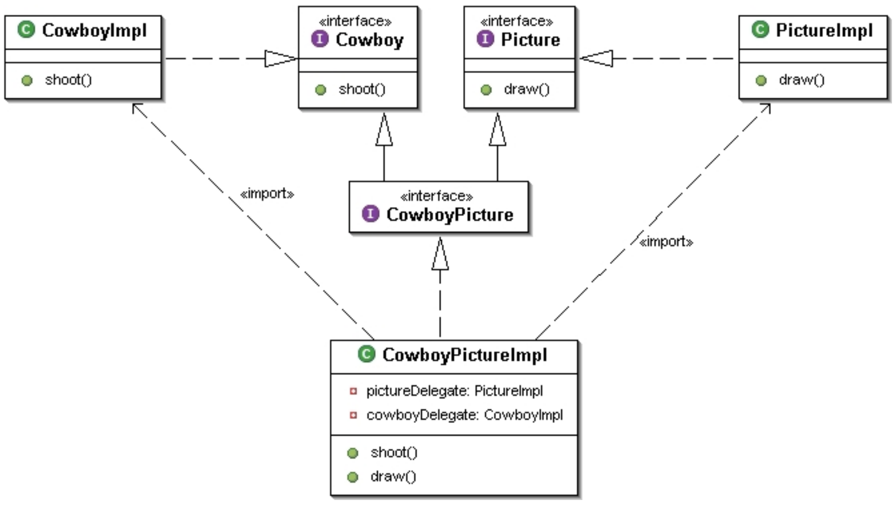

Figure 2.2: Multiple inheritance in Java.

The object referenced by cowboyPicture is of type: **CowboyPictureImpl**, **CowboyPicture**, **Cowboy**, **Picture**, **Any** and **Object** and a call to draw() respectively shoot() will execute the correct code sequences. As one can see modelling multiple inheritance is quite easy that way but this is only possible if the target language supports multiple subtyping.

2.2.4 Renaming

A difficulty when mapping Eiffel to Java is Eiffel's ability to rename a feature in a subclass. The problem is Java inflexibility: A method with the same signature as a method in the superclass always overrides this method. In .Net it is possible to hide a method of a superclass and prevent a dynamic binding by declaring the method as non virtual. To illustrate the problem an important command is added to *COWBOY*: *draw*. Before a cowboy can shoot he has to draw his weapon:

```
class COWBOY
feature
        shoot is do σ1 end
        draw is do σ2 end
end
```

Compiling this system will result in a compile time error:

Error code: VMFN
Error: two or more features have same name.
What to do: if they must indeed be different features, choose different
names or use renaming; if not, arrange for a join (between deferred
features), an effecting (of deferred by effective), or a redefinition.

Class: *COWBOY_PICTURE*
Feature: draw inherited from: *PICTURE* Version from: *PICTURE*
Feature: draw inherited from: *COWBOY* Version from: *COWBOY*

Because *COWBOY_PICTURE* does inherit two different feature with the same name: The *draw* from *PICTURE* to draw a picture to a surface and the *draw* from *COWBOY* to draw the cowboy's weapon. These are two completely different features and they must be distinguished somehow in *COWBOY_PICTURE*. Eiffel provides a construct which allows to do that: Renaming.

```
class COWBOY_PICTURE
inherit
        COWBOY rename draw as draw_weapon end
        PICTURE
end
```

This means, that the feature *draw* from *COWBOY* is now accessed in *COWBOY_PICTURE* through a call on *draw_weapon* and a call to *draw* in *COWBOY_PICTURE* will yield σ_0 and not σ_2:

```
                local
                        cowboy_picture: COWBOY_PICTURE
                do
                        create cowboy_picture
                        cowboy_picture.draw          -- yields σ0
                        cowboy_picture.draw_weapon   -- yields σ2
                        cowboy_picture.shoot         -- yields σ1
                end
```

The Java translation:

```
public interface Cowboy extends Any {
    public void shoot();
    public void draw();
}


```

```java
public class CowboyImpl implements Cowboy {
    public void shoot() {σ1}
    public void draw() {σ2}
}

public interface CowboyPicture extends Cowboy, Picture {
    public void drawWeapon();
}

public class CowboyPictureImpl implements CowboyPicture {

    private PictureImpl pictureDelegate = new PictureImpl();
    private CowboyImpl cowboyDelegate = new CowboyImpl();

    public void shoot() {cowboyDelegate.shoot();}
    public void draw() {pictureDelegate.draw();}
    public void drawWeapon() {cowboyDelegate.draw();}
}

        CowboyPicture cowboyPicture = new CowboyPictureImpl();
        cowboyPicture.draw();        -- yields σ0
        cowboyPicture.drawWeapon();  -- yields σ2
        cowboyPicture.shoot();       -- yields σ1
```

That works fine so far. But if we put the object referenced by *cowboy_picture* into an *ARRAY* of *COWBOY's* and call then *draw* on the object in the *ARRAY* will the *COWBOY_PICTURE* object be drawn to a surface or will the cowboy draw its weapon?[b]

```eiffel
1          local
2                  cowboy_picture: COWBOY_PICTURE
3                  cowboys: ARRAY[COWBOY]
4          do
5                  create cowboy_picture
6                  cowboy_picture.draw      -- yields σ0
7
8                  create cowboys.make (0, 0)
9                  cowboys.put (cowboy_picture, 0)
10                 cowboys.item (0).draw    -- yields σ2
11         end
```

Something very interesting happens here: Although the call to *draw* in line 6 and 10 are invoked on the exact same object the executed code sequences are not the same. In Eiffel not only the type of an object at runtime is taken into consideration when a binding to an implementation is done but also the static type of the expression which returns the reference to that object does matter in some cases. Here the type of *cowboy_picture* expression is *COWBOY_PICTURE* but the type of *cowboys.item (0)* is *COWBOY* and therefore on line 11 *draw* from *COWBOY* is called.

[b]For a cowboy the difference is important for obvious reasons.

Now let us try to do the same with Java:

```
1         CowboyPicture cowboyPicture = new CowboyPictureImpl();
2         cowboyPicture.draw();   //yields σ0
3
4         Cowboy[] cowboys = new Cowboy[1];
5         cowboys[0] = cowboyPicture;
6         cowboys[0].draw();     //yields σ0
```

Not surprisingly line 6 does not do the right thing. In Java all methods are virtual and the runtime does always bind to the most recent implementation. Why should it not? The runtime does not know that draw() from **Cowboy** is a completely different method then draw() from **Picture**. Unfortunately there is no way to convince Java that these are different methods.
There is even a worse case: Adding a query *size* to *COWBOY* returning the size of a cowboy in meters and adding a query *size* to *PICTURE* returning the size of a picture in bytes.

```
class COWBOY
feature
        size: DOUBLE is do σ3 end
end

class PICTURE
feature
        size: INTEGER is do σ4 end
end

class COWBOY_PICTURE
inherit
        COWBOY rename size as cowboy_size end
        PICTURE
end
```

Generating valid Java code with the ad hoc pattern is not possible[c]:

```
public interface Cowboy {
    public Double size();
}

public interface Picture {
    public Integer size();
}

public interface CowboyPicture extends Cowboy, Picture {}
```

CowboyPicture is not a valid Java interface because the signatures of the two size() queries are not the same.
It follows a list of possible solutions to the problem:

1. Name mangling.
   Since draw() in **Cowboy** denotes another method then draw() in **Picture** another name is given to the two: Cowboy_draw() and Picture_draw(). In line 6 cowboys[0].Cowboy_draw()

[c]It's also clear now why Eiffel has to use static binding in this case. Besides the intuition which the static binding solution has, it is also required to guarantee type safety.

is called since the type of the expression cowboys[0] is **Cowboy** and σ_2 is executed.

```
public class CowboyPictureImpl implements CowboyPicture {

    public void Picture_draw() {
            pictureDelegate.Picture_draw();
    }
    public void Cowboy_draw() {
            cowboyDelegate.Cowboy_draw();
    }
    public Integer Picture_size() {
            return pictureDelegate.Picture_size();
    }
    public Double Cowboy_size() {
            return cowboyDelegate.Cowboy_size();
    }
}
```

Pros

- Fast

Cons

- No one-to-one mapping of Eiffel feature names to Java method names.
- Easy for the user to make a mistake.

2. Hide name mangling behind static method.
   One can hide the name mangling behind a static method with the correct Eiffel name and call in this method the correct implementation.

```
public class CowboyPictureImpl implements CowboyPicture {
    ...
    public static void draw(CowboyPicture object) {
        object.Picture_draw();
    }
    public static void drawWeapon(CowboyPicture object) {
        object.Cowboy_draw();
    }
    public static Integer size(CowboyPicture object) {
        return object.Picture_size();
    }
    public static Double cowboySize(CowboyPicture object) {
        return object.Cowboy_size();
    }
}

        CowboyPictureImpl.draw(cowboyPicture);
        CowboyImpl.draw(cowboys[0]);
```

Pros

- Fast
- No visible name mangling.

Cons

- Very strange way of object oriented programming.
- Easy for the user to make a mistake.

3. Static type as parameter.
The type of an expression is passed as a parameter to every method:

```
    cowboys[0].draw(Cowboy.class);
```

draw() from **CowboyPictureImpl** can then decide which implementation to call:

```
    public void draw(Class staticType) {
        if (isSupertype(staticType, Cowboy.class) {
            cowboyDelegate.draw (staticType)
        } else {
            pictureDelegate.draw (staticType);
        }
    }
```

Now the call on cowboys[0] will yield σ_2.
Pros

- Easy to implement.
- No name mangling.

Cons

- Slow.
- Additional parameter for every method.
- Does not solve different return types problem.

4. Overloading.
Overloading allows to have a faster and nicer implementation of the static type as parameter approach. Java does allow to have different routines with the same name in one class, through overloading.

```
public interface Cowboy {
    public void draw(Cowboy self);
}
```

A parameter called self with the same type of the interface is added to every routine. Now the **CowboyPicture interface** looks a little bit different:

```
public interface CowboyPicture extends Cowboy, Picture {
    //Rename draw as draw_weapon
    public void drawWeapon (CowboyPicture self);
    public void draw (CowboyPicture self);
}

public class CowboyPictureImpl implements CowboyPicture {
    public void draw (Cowboy self)         {cowboyDelegate.draw(self);}
    public void draw (Picture self)        {pictureDelegate.draw(self);}
    public void draw (CowboyPicture self) {pictureDelegate.draw(self);}
    public void drawWeapon (CowboyPicture self) {
        cowboyDelegate.draw(self); }
}
```

A client can use the system like following:

```
        cowboyPicture.draw(cowboyPicture); //yields σ0
        cowboys[0].draw(cowboys[0]);       //yields σ2
```

In line 4 the correct implementation is now executed: The one in **Cowboy**
Pros

- Easy to implement.
- No name mangling.

Cons

- Additional parameter for every method.
- Expressions like a.f.g are translated to a.f(a).g(a.f(a)) which is not only slow but also not correct if a.f has any side effects.

5. Reflection.
The overloading solution implies that the caller has to know the type of every expression, otherwise the runtime wouldn't be able to bind to the correct implementation. It's possible to read out the stack trace in the callee[d] and find the exact line number where the call was invoked in the caller. Then the callee can parse the callers class and read out the static type of the expression from the constant pool.
Pros

   - No name mangling.
   - No additional parameter.

   Cons

   - Does not solve the incompatible return type problem.
   - Way too slow.
   - Difficult to implement.

6. Real casting.
Instead of distinguishing the generated routines somehow (through overloading or name mangling) it would also be possible to call the method draw() on different objects. A cast is only a type check at runtime. If the type of the object referenced by the element on top of the execution stack does not conform to the required type an exception is thrown, otherwise the stack remains unchanged. If the reference could be changed to another object then the call could be bound to another implementation. For this purpose queries are added to all interfaces which return objects which do not only have the type of a parent but which are instances of the parents implementing class:

```
public interface CowboyPicture extends Cowboy, Picture {
    ...
    public Cowboy asCowboy();
    public Picture asPicture();
}

public class CowboyPictureImpl implements CowboyPicture {
    private PictureImpl pictureDelegate = new PictureImpl();
    private CowboyImpl cowboyDelegate = new CowboyImpl();
```

[d] (new Throwable).getStackTrace()

```
10        ...
11        public Cowboy asCowboy() {return cowboyDelegate;}
12        public Picture asPicture() {return pictureDelegate;}
13    }
14
15        cowboyPicture.draw();
16        cowboys[0] = cowboyPicture.asCowboy();
17        cowboys[0].draw();
```

Now the two calls on line 15 and 17 are not invoked on the same object anymore and the correct implementation is executed. It is also possible to make a downcast back from cowboys[0] to a **CowboyPicture** by adding reference in **CowboyImpl** to a **CowboyPictureImpl**. One of the problems with this solution is, that now there is no polymorphism anymore. What if[e] *COWBOY_PICTURE* does redefine *shoot* from *COWBOY*? A call on cowboys[0] to shoot() will then execute the version from *COWBOY*, but the redefined version from *COWBOY_PICTURE* should be executed. There is a dirty solution for that problem: Generating an other **CowboyImpl** class, one that is only there for **CowboyPictureImpl** and which contains the new implementation:

```
public class CowboyForCowboyPictureImpl implements Cowboy {
    ...
    public void shoot() {
        //Redefined implementation from COWBOY_PICTURE.
    }
}
```

The cowboyDelegate in **CowboyPictureImpl** is then a reference to a instance of **CowboyForCowboyPictureImpl**.
Pros

- No name mangling.
- Fast.

Cons

- Does not solve the incompatible return type problem.
- Number of generated classes may be very huge.
- Difficult to implement.

7. Real casting without subtyping.
The only solution for the incompatible return type problem when using real casting is to get rid of explicit subtyping relations. The **CowboyPicture** interface does not extend **Cowboy** and **Picture**. The subtyping structure is still there but not explicit.
Pros

- No name mangling.
- Fast.

Cons

- No explicit subtyping relations.
- The user will not see the subtyping relations in an IDE anymore.

[e]For some obscure reason.

The overloading approach is a solution under the assumption that an expression like a.b.c can be translated to:

```
        TypeOfgetB tmp = a.getB(a);
        tmp.c(tmp);
```

The two expressions are equivalent as long as the return type of getB() does not change. Changing the type would require to change the call as well. But this also holds for name mangling:

```
        a.TypeOfa_getB().TypeOfgetB_c();
```

Changing the type of TypeOfa_getB would require to change the call to c. The real casting without subtyping is a solution, but one of the goals is to make the subtyping structure visible to Java programmers.
The overloading and name mangling approach are indeed very similar. Overloading just moves the mangling from the routine name to its signature. From now on only the name mangling approach is discussed.

2.2.5 Name mangling

To make it easier for a Java programmer to select the right implementation of a feature all routine names are prefixed with the name of the interface. The user then calls the routine prefixed with the name of the class which is the type of the expression on which the call is made. *COWBOY_PICTURE* is translated to Java like following:

```
public interface Cowboy extends Any {
     public void Cowboy_draw();
}

public interface Picture extends Any {
     public void Picture_draw();
}

public interface CowboyPicture extends Cowboy, Picture {
    public void CowboyPicture_draw();
    public void CowboyPicture_drawWeapon();
}

public class CowboyPictureImpl implements CowboyPicture {
    private Cowboy cowboyDelegate;
    private Picture pictureDelegate;

    public void CowboyPicture_draw()       {draw();}
    public void Picture_draw()             {draw();}
    public void CowboyPicture_drawWeapon() {drawWeapon();}
    public void Cowboy_draw()              {drawWeapon();}

    protected void draw() {pictureDelegate.Picture_draw();}
    protected void drawWeapon() {cowboyDelegate.Cowboy_draw();}

}
```

The following example shows how a Java programmer can use this system as a client:

```java
public static void main(String[] args) {
    CowboyPicture cp = new CowboyPictureImpl();
    cp.CowboyPicture_draw();
    cp.CowboyPicture_drawWeapon();
    Cowboy c = cp;
    c.Cowboy_draw();
    Picture p = cp;
    p.Picture_draw();
}
```

The following example shows how to use it to extend **CowboyPictureImpl**:

```java
public class MyCowboyPicture extends CowboyPictureImpl {
    protected void draw() {
           super.draw();
           //Some other code in Java here
    }
}
```

This leads to the first translation rule:

Rule 1: ∀ f element routines of Eiffel class F: Method F_{name}_f_{name} is added to the interface F_{name} and a protected method f_{name} is added to the class F_{name}Impl. F_{name}_f_{name} in F_{name}Impl calls f_{name} and f_{name} calls either f from a direct parent Impl class or contains the translated code if f is new, redefined or effected in F.

2.2.6 Constructors

What if **AImpl** does not have a default constructor? What if it has more then one constructor? How will **BImpl** instantiate the delegate object?
Let us take the following Eiffel classes as example:

```eiffel
class A
create make_a
feature
        make_a is do σ0 end
end

class B
inherit A
create make_a, make_b
feature
        make_b is do σ1 end
end

        local
                b: B
        do
                create b.make_a
                create b.make_b
        end
```

Another problem is that Java does not allow to have named constructors. Constructors have always the name of the class, the only way to have multiple constructors is through overloading. The trick is to provide a default constructor for all Impl classes and to add methods for every creation feature in Eiffel to the Impl classes. Also introduced is a new set of classes: Constructor. This set of classes have a static function for every constructor in the Eiffel class each returning an instance of the Impl class[f]. The user can use one of the static functions to get a new instance and has therefore never to care about the Impl classes. This way the information hiding is still good enough. The Constructor classes are pretty simple and will look like this:

```
public class AConstructor {

    public static A makeA() {
        AImpl res = new AImpl();
        res.A_makeA();
        return res;
    }
}

public class BConstructor {

    public static B makeA() {
        BImpl res = new BImpl();
        res.B_makeA();
        return res;
    }

    public static B makeB() {
        BImpl res = new BImpl();
        res.B_makeB();
        return res;
    }
}

    A a;
    a = AConstructor.makeA();
    B b;
    b = BConstructor.makeA();
    b = BConstructor.makeB();
```

This requires a new rule:

Rule 2: ∀ c element constructors of Eiffel class F: Add **public static** F_{name} c_{name} to F_{name}Constructor which returns a new instance of F_{name}Impl on which $F_{name}\_c_{name}$ was called.

Other than that, a constructor is treated the same way as any other feature.

2.2.7 Sharing

Since Eiffel does support multiple inheritance the subclassing graph is not a tree but a Directed Acyclic Graph. A diamond structure can be easily built:

[f] A.k.a. a factory

```
class F
feature
        f is do σ0 end
end

class A inherit F end
class B inherit F end
class C inherit A, B end
```

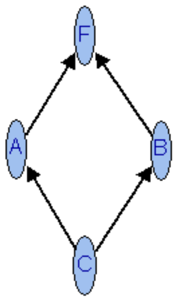

Figure 2.3: Diamond inheritance structure in Eiffel

In this case *C* inherits twice the feature *f*: From *A* and *B*. But since none of the parent classes did redefine or rename *f*, *f* from both direct parents are exactly the same and it therefore doesn't matter to which parent the call on *f* in *C* is delegated:

```
public class CImpl implements C {
    public void C_f() {f();}

    protected void f() {aDelegate.A_f();}
}
```

In the example above it really doesn't matter if the call on C_f() is delegated to A_f() or B_f(). But if *B* or *A* do not directly inherit from *F* it does because of performance reasons. The call should then be delegated to the implementing class which is closer to the definition of *f* since there are less indirections on this path.

Rule 3: If there is more than one parent method to which a call can be delegated the parent method which is closest to the definer of the method is chosen. A method f is closer than a method g if the number of nodes between the class in which f is a member and the class defining f is less than the number of nodes between the class in which g is a member and the class defining g.

2.2.8 Join with select

What happens if *f* is redefined in *A*?

```
class A
inherit F redefine f end
feature
        f is do σ1 end
end
```

Now let us take a simple command:

```
        do_f(object: F) is
                do
                        object.f
                end
```

Which *f* should now be executed if object is of type *C*? There are two possibilities: The one as defined in *F* since *C* inherits this feature through *B* or the one as defined in *A*. Another problem is that now there are two different *f*s in *C*. This problem can be solved with renaming. To solve the first problem Eiffel provides the **select** construct:

```
class C
inherit
        A rename f as af select af end
        B
end
```

This means that if an expression has type *C* a call on *f* will execute the *f* as implemented in *F* and a call on *af* will execute the *f* as implemented in *A*. But if an expression has type *A*, *B* or *F* the *f* as defined in *A* is executed, even tough if the dynamic type of the object is *C*[g]:

```
1          local
2                  c: C
3                  f: F
4          do
5                  create c
6                  c.f     -- yields σ0
7                  c.af    -- yields σ1
8                  f := c
9                  f.f     -- yields σ1
10         end
```

The same calls on the same objects in line 6 and 9 execute different code sequences:

```
public interface C extends A, B {
    //A rename f as af select af end
    public void C_f();
    public void C_af();
}

public class CImpl implements C {

    public void F_f()  {af();}
    public void A_f()  {af();}
    public void B_f()  {af();}
    public void C_f()  {f();}
    public void C_af() {af();}

    protected void f()  {bDelegate.B_f();}
    protected void af() {aDelegate.A_f();}
}

```

[g]Note the inconsistency here: *f* from *A* is executed if an expression of type *B* returns an object of type *C*

```
    C c = CConstructor.defaultCreate();
    c.C_f();   //yields σ0
    c.C_af(); //yields σ1
    F f = c;
    f.F_f();   //yields σ1
```

Rule 4: An unselected feature is treaded as if it is a new feature.

2.2.9 Attributes

Until now the assumption was made that the different code sequences do not change the state of an object. This does of course not hold for most of the routines in a class. Let us take the following Eiffel example:

```
class A
feature
        i: INTEGER
        inc is do i := i + 1 end
end

class B
inherit A
feature
        dec is do i := i − 1 end
end
```

In this example *B* does change the value of an attribute in **A** which is no problem in Eiffel, since an **inherit** is a subclassing and **B** does have access to *i* as if it where its own attribute. Normally this is also the case in Java[h]. But the mapping pattern only conserves the subtyping structure. There is no subclassing anymore. **BImpl** has no access to the field i in **AImpl** unless i is made public in **AImpl**. If i is made public than any other class has write access on i whereas in Eiffel only subtypes of *A* do have write access on *i*. But this is not too much of a problem, since the user defines its variables as of type **A** and not **AImpl** and will therefore not see the field i if it is not added to the interface **A**. In **A** a public function getI() which acts the same way as the feature *i* in *A* is defined[i]. In **AImpl** a **private** attribute i and a **public** command setI_ to allow **BImpl** to change the state of **AImpl** is defined.

```
public interface A extends Any {
    public Integer A_getI();
    public void A_inc();
}

public interface B extends A {
    public void B_dec();
    public void B_inc();
    public Integer B_getI();
}
```

[h]If the field is visible to a subclass

[i]The compiler will have an option to call the function i() to come closer to the Eiffel style, but since getI() is the normal Java style this will be the default.

```
public class AImpl implements A {
    private Integer i;

    //Visible to clients of A
    public Integer A_getI() {return getI();}
    public void A_inc() {inc();}

    //Visible to clients of AImpl ("subclasses")
    public void A_setI_(Integer i) {setI_(i);}

    protected void setI_(Integer i) {this.i = i;}
    protected void inc() {i++;}
    protected Integer getI() {return i;}
}

public class BImpl implements B {
    private AImpl aDelegate;

    //Visible to clients of B
    public Integer A_getI() {return getI();}
    public void A_inc() {inc();}
    public Integer B_getI() {return getI();}
    public void B_inc() {inc();}
    public void B_dec() {dec();}

    //Visible to clients of BImpl
    public void B_setI_(Integer i) {setI_(i);}

    protected void setI_(Integer i) {aDelegate.A_setI_(i);}
    protected void inc() {aDelegate.A_inc();}
    protected Integer getI() {return aDelegate.A_getI();}
    protected void dec() {setI_(getI() - 1);}
}
```

This leads to the second rule:

Rule 5: ∀ a element attributes of Eiffel class F: Add feature get_a$_{name}$: a$_{type}$ exported to *ANY* and feature set_a$_{name}$_(a$_{name}$: a$_{type}$) exported to *NONE* to F. If a is new add field private a$_{type}$ a$_{name}$ to F$_{name}$Impl.

Attributes are handled as if there where two new features in the Eiffel class: A public getter and a private setter feature. How to handle private features is discussed in subsection 2.2.17. This also works in combination with renaming:

```
class B
inherit A rename i as j end
feature
        dec is do j := j - 1 end
end
```

```
public interface B extends A {
    //Rename i as j
    public Integer B_getJ();
    public void B_dec();
    public void B_inc();
}

public class BImpl implements B {
    public Integer A_getI() {return getJ();}
    public void A_inc() {inc();}

    public Integer B_getJ() {return getJ();}
    public void B_inc() {inc();}
    public void B_dec() {dec();}

    public void B_setJ_(Integer j) {setJ_(j);}

    protected void setJ_(Integer j) {aDelegate.A_setI_(j);}
    protected void inc() {aDelegate.A_inc();}
    protected Integer getJ() {return aDelegate.A_getI();}
    protected void dec() {setJ_(getJ() - 1);}
}
```

2.2.10 Sharing of attributes

Unfortunately the approach presented in the previous section wont work in every case since not only features but also attributes can be shared. Let us make another example:

```
class A
feature
      i: INTEGER
end

class B
inherit A
feature
      inc is do i := i + 1 end
end

class C
inherit A
feature
     dec is do i := i - 1 end
end

class D inherit B,C end

     local
         d: D
     do
         create d
         d.inc; d.inc; d.dec
```

```
25          print (d.i.out)
26      end
```

The code on line 25 will print 1 to the console because *inc* and *dec* operate both on the same attribute: *i* is shared among *A*, *B*, *C* and *D*. But if the code is translated to Java then **BImpl** will use another instance of **AImpl** as **CImpl** and inc will therefore operate on another attribute then dec. One solution of the problem would be that **BImpl** and **CImpl** uses the same instance of **AImpl** as delegation object. But this would require that **DImpl** passes this instance to **BImpl** and **CImpl** on creation, to be precise **DImpl** would have to pass an instance of every implementing class of every superclass, direct or not, meaning in this example, also an instance of **AnyImpl**. Fortunately there is a much simpler solution: Another hierarchy of interfaces and classes is generated which only stores attributes for every implementing class. The effect is that the code of a class is completely separated from its data:

```java
interface AAttrs {
        public void setI(int i);
        public int getI();
}

class AImpl implements A {
        private AAttrs attrs;

        public AImpl (AAttrs attrs) {this.attrs = attrs;}

        protected int getI() {return attrs.getI();}
        protected void setI_(int i) {attrs.setI(i);}
}

interface BAttrs extends AAttrs {}

class BImpl implements B {
        private BAttrs attrs;
        private AImpl aDelegate;

        public BImpl (BAttrs attrs) {
                this.attrs = attrs;
                aDelegate = new AImpl (attrs);
        }
        ...
        //getI and setI_ delegate calls
        //to an instance of AImpl as before
}

interface CAttrs extends AAttrs {}
interface DAttrs extends BAttrs, CAttrs {}

class DAttrsImpl implements DAttrs {
        private int i;

        public void setI(int i) {this.i = i;}
        public int getI() {return i;}
}
```

```
class DImpl implements D {
      private DAttrs attrs;
      private BImpl bDelegate;
      private CImpl cDelegate;

      public DImpl() {
            this (new DAttrsImpl());
      }

      public DImpl (DAttrs attrs) {
            this.attrs = new DAttrsImpl();
            bDelegate = new BImpl (attrs);
            cDelegate = new CImpl (attrs);
      }
      ...
}
```

A new constructor is defined in every class and an additional field called attrs. The parameter of the constructor has the same type as the field. The field is set to reference to the object passed as parameter to the constructor. This object is passed up to every delegation object. In the end all the instance of all the implementing class in a subtype hierarchy share the same attrs object. If a class introduces a new attribute then a setter and getter is added to its Attrs interface. The implementing class of the Attrs interface has to implement all this attributes. The name of the attributes in the Attrs interfaces does not matter, they just have to be unique. Here the name of the attribute is i for simplicity. But the code generator will just give it a unique name, starting with an a followed by a unique number. The class diagram on page 44 shows the translated system (without the *inc* and *dec* commands and without **Any**).

Rule 6: ∀ classes F: An interface F_{name}Attrs containing setter and getter methods for every new attribute in F and a class F_{name}AttrsImpl implementing F_{name}Attrs are added to the system.

Rule 7: A constructor with a parameter of type F_{name}Attrs is added to every class F_{name}Impl and the default constructor calls this constructor with a new instance of F_{name}AttrsImpl.

2.2.11 Inheriting multiple times from the same class

In Eiffel it is possible to inherit multiple times from the same class:

```
class A
feature
      i: INTEGER
      inc is do i := i + 1 end
end

class B
inherit
      A select i, inc end
      A rename i as j, inc as incj end
end
```

Interesting here is that *incj* does not operate on *j* but on *i* since *i* is the selected one. This may be confusing but it is consequent because *i* is shared among *A* and *B*. To make it clearer the system can be translated to the following system without changing its semantic (class *A* remains unchanged):

```
class C inherit A end

class D
inherit
    A rename i as j, inc as incj end
end

class E
inherit
    C select i, inc end
    D
end
```

Features *i*, *j*, *inc* and *incj* do have the exact same semantics in class *B* and *E*. How to translate *E* is explained in the previous two sections. There is not much special to the translation of class *B*. The compiler just has to make sure, that it does not generate two delegates with the same name:

```
interface AAttrs extends AnyAttrs {
    public void set1(int a1);
    public int get1();
}

class AImpl implements A {
    ...
    protected void inc() {setI_ (getI() + 1);}

    protected void setI_ (int i) {attrs.set1(i);}
    protected int getI() {return attrs.get1();}
}

interface BAttrs extends AAttrs {
    public void set2(int a2);
    public int get2();
}

class BImpl implements B {

    private AImpl aDelegate;
    private BAttrs attrs;

    public BImpl (BAttrs attrs) {
        this.attrs = attrs;
        aDelegate = new AImpl (attrs);
    }

    protected void inc() {aDelegate.A_inc();}
    protected void incj() {aDelegate.A_inc();}
    protected int getI() {aDelegate.A_getI();}
    protected int getJ() {return attrs.get2();}
    protected void setJ(int j) {attrs.set2(j);}
    ...
}
```

This conforms to rule 4: Unselected features are treaded as if they where new.

2.2.12 Constants

It is possible to define an attribute as constant in Eiffel:

```
        i: INTEGER is 100
```

Since it is guaranteed that the value denoted by *i* is always 100 every occurrence of *i* can be replaced by the value of *i*. For example:

```
        j := i + k
```

This assignment is translated to:

```
        setJ_ (100 + getK());
```

2.2.13 Redefining and Precursors

A redefine will lead to the generation of a new implementation instead of the generation of a call to an other Impl class. A **Precursor** will generate a call the implementation of the parent implementation at the position of the precursor:

```
class A
feature
        i: INTEGER
        inc is do i := i + 1 end
end

class B
inherit A redefine inc end
feature
        inc is
                do
                        σ0
                        Precursor{A}
                        σ1
                end
end
```

This is translated to:

```
public interface A {public void A_inc();}

public class AImpl implements A {
    public void A_inc() {inc();}
    protected void inc() {setI (getI() + 1);}
}

public interface B extends A {public void B_inc();}
public class BImpl implements B {

    public void A_inc() {inc();}
    public void B_inc() {inc();}

    protected void inc() {
        σ0;
        aDelegate.A_inc(); //Precursor{A}
        σ1;
    }
}
```

Rule 8: ∀ p element precursors in feature f in F: Call f from parent Impl class at position of p.

2.2.14 Deferred features

Java does know the notion of deferred, it's just called abstract, but the concept is exactly the same. The problem is that defining a method in a class as abstract leads to the requirement of defining the class as abstract and the class can therefore not be instantiated. But other implementing classes may delegate calls to this class and need an instance of the abstract class. Effecting all deferred features in the Impl classes with an empty method body is one possibility, but a Java programmer extending the Impl class does not know which routines to implement and which are already implemented. That's why another set of classes is introduced: ImplConcrete classes. These classes extend an abstract Impl class and do add implementations for every abstract method in the Impl class: Implementations with an empty body. These methods are never called.

```
deferred class A
feature
     f is deferred end
     g is do σ0 end
end

class B
inherit A
feature
    f is do σ1 end
end
```

```
public interface A extends Any {
     public void A_f();
     public void A_g();
}

public abstract class AImpl implements A {
     public void A_g() {g();}
     public void A_f() {f();}
     protected void g() {σ0;}
     protected abstract void f();
}


```

```
public class AImplConcrete extends AImpl {
      protected void f() {//Never called};
}

public interface B extends A {
      public void B_f();
      public void B_g();
}

public class BImpl implements B {
      private AImpl aDelegate = new AImplConcrete();

      public void A_f() {f();}
      public void A_g() {g();}
      public void B_f() {f();}
      public void B_g() {g();}

      protected void f() {σ1;}
      protected void g() {aDelegate.A_g();}
}
```

This works fine as long as the user does not instantiate AImplConcrete himself, but uses the instance which is returned by one of the factory functions from the Constructor class.

Rule 9: If Eiffel class F is deferred then F_{name}Impl is abstract, a java class F_{name}ImplConcrete extending F_{name}Impl is generated and delegaters to F_{name}Impl use an instance of F_{name}ImplConcrete.

Rule 10: For all f element deferred features in F: f_{name} is abstract in F_{name}Impl and f_{name} with an empty body is added to F_{name}ImplConcrete.

2.2.15 Undefine

An undefine statement takes an effective feature of a parent class and turns it into a deferred feature. The translation is straight forward: An undefined feature is handled in the exactly same way as a deferred feature. And in fact such a feature is nothing else than a deferred one.

2.2.16 Covariance

Covariance the ability of Eiffel to change the type of a query or the types of arguments of a routine in a subclass to a subtype of the type as defined in the parent class. An example can be found in Object-Oriented Software Construction [6, page 622]:

```
class SKIER
feature
        roommate: like Current
        share (other: like Current) is
                do
                        roommate := other;
                end
end
```

```

class BOY inherit SKIER end
class GIRL inherit SKIER end
```

This system should express, that a *BOY* can not share a room with a *GIRL*. Let us try to translate that system to Java:

```
public interface Skier extends Any {
    public Skier Skier_getRoommate();
    public void Skier_share(Skier other);
}

public class SkierImpl implements Skier {
    public Skier Skier_getRoommate()              {return getRoommate();}
    public void Skier_setRoommate_ (Skier other) {setRoommate_(other);}
    public void Skier_share(Skier other)          {share(other);}

    protected void share(Skier other)          {setRoommate(other);}
    protected Skier getRoommate()              {return attrs.get1();}
    protected void setRoommate_(Skier other) {attrs.set1(other);}
}
```

A setter and getter method is generated for the attribute roommate and the command share(). Now let us translate *BOY*.

```
public interface Boy extends Skier {
    public Boy Boy_getRoommate();
    public void Boy_share(Boy other);
}

public class BoyImpl implements Boy {
    private SkierImpl skierDelegate = new SkierImpl();

    public Boy Boy_getRoommate()     {return getRoommate();}
    public Skier Skier_getRoommate() {return getRoommate();}

    public void Boy_setRoommate_ (Boy other) {setRoommate_ (other);}

    public void Boy_share(Boy other) {share(other);}
    public void Skier_share(Skier other) {
        share (other);
        //share ((Boy) other);
    }

    protected void share(Boy other) {
        skierDelegate.Skier_share(other);
    }

    protected Boy getRoommate() {
        return skierDelegate.Skier_getRoommate();
        //return (Boy) skierDelegate.Skier_getRoommate();
    }


30      protected void setRoommate_ (Boy other) {
31          skierDelegate.Skier_setRoommate_ (other);
32      }
33  }
```

The translation of *GIRL* is equivalent, except that **Boy** is replaced by **Girl**. There are two remarkable things about this class: 1. **BoyImpl** is a very long class considering that *BOY* consists of only 5 words. 2. The program will not compile since it is not type safe: On line 15 a **Skier** is passed to share which expects at least a **Boy** and on line 24 getRoommate has to return a **Boy** but Skier_getRoommate only returns a **Skier**. One would expect that casts as shown in the commented code on line 16 and 21 are required but in fact it is possible to generate Java Byte Code without the casts that is not rejected by the verifier:

```
Method public  Skier_share (Skier ) −> void
0       aload_0
1       aload_1
2       invokenonvirtual #36 <Method BoyImpl.share (LBoy;)V>
5       return

Method protected  getRoommate () −> Boy
0       aload_0
1       getfield #17 <Field BoyImpl.skierDelegate LSkierImpl;>
4       invokevirtual #66 <Method SkierImpl.Skier_getRoommate ()LSkier;>
7       areturn
```

The two methods above are accepted by the Java Byte Code verifier. It is even possible to execute the following Java program without a crash at runtime:

```
public static void main(String[] args) {
    Boy b = new BoyImpl();
    Girl g = new GirlImpl();
//  b.Boy_share(g); rejected
    Skier s = b;
    s.Skier_share(g);
}
```

This is exactly the same behaviour as in Eiffel. It is remarkable that it is possible to map Eiffel's covariance to Java Byte Code. Of course it is not possible to implement covariance in Java. But on the Byte Code level covariance is no problem as long as one uses interfaces as parameter respectively as return types.

Rule 11: $\forall$ l element like in F: Replace "like l_{name}" by l_{type}

2.2.17 Export statement

With Eiffel's **export** statement one can either hide a feature of a parent class in a subclass, or export a hidden feature from a superclass in a subclass.
Let's look at the second option first:

```
class A
feature {NONE}
        f is do σ₀ end
end
```

```
class B
inherit A export {ANY} f end
end
```

An expression of type *A* does not have access to *f* but an expression of type *B* can call *f*. Until now the assumption was that all features in Eiffel classes are public but of course this is not the case in reality. Fortunately translating the above example is straight forward: A_f is not added to the Java interface **A** but to **AImpl**. This way a client of **A** does not have access to A_f, but **BImpl** has access since **BImpl** defines its delegation object as of type **AImpl**.

```
public interface A extends Any {}

public interface B extends A {public void B_f();}

public class AImpl implements A {
    public void A_f() {f();}
    protected void f() {σ0;}
}

public class BImpl implements B {
    private AImpl aDelegate = new AImpl();

    public void B_f() {f();}
    protected void f() {aDelegate.A_f();}
}
```

Rule 12: ∀ f element features exported to NONE in F: Do not generate a public routine F_{name}_f_{name} in F_{name} but one in F_{name}Impl.

Let us look at descendant hiding: This means that a feature of a superclass can be hidden in a subclass. There is an other example in Object-Oriented Software Construction [6, page 262f]:

```
class POLYGON
feature
        vertex_count: INTEGER
        add_vertex is
                do
                        vertex_count := vertex_count + 1;
                end
end

class RECTANGLE
inherit
        POLYGON export {NONE} add_vertex end
end
```

Here a rectangle is a polygon with 4 sides. That's what the system tries to model. The problem is, that it's possible to add a vertex to a polygon but not to a rectangle. That's why *add_vertex* from class *POLYGON* is hidden in *RECTANGLE*. But there is an easy way to destroy the invariant of a rectangle object:

```
        local
                p: POLYGON
                r: RECTANGLE
                b: BOOLEAN
        do
                create r
        --      r.add_vertex not possible
                p := r
                p.add_vertex
        end
```

Let us translate *RECTANGLE* to Java:

```
public interface Rectangle extends Polygon {
    public Integer Rectangle_getVertexCount();
}

public class RectangleImpl implements Rectangle {
    public Integer Rectangle_getVertexCount() {return getVertexCount();}
    public Integer Polygon_getVertexCount() {return getVertexCount();}
    public void Polygon_addVertex() {addVertex();}

    protected Integer getVertexCount() {
          return polygonDelegate.Polygon_getVertexCount();
    }
    protected void addVertex() {
          throw new Error("System␣validity␣error:␣Call␣to␣hidden␣feature.");
    }
}
```

Rule 12 states that Rectangle_addVertex() is not added to **Rectangle**. In addition calls to Polygon_addVertex() on a **Rectangle** object leads to a runtime error.[j]

Rule 13: ∀ f element features exported to NONE in F: If f is inherited and was not exported to NONE then throw a system validity error in f_{name} in F_{name}Impl.

2.2.18 Friends

Besides exporting a feature to NONE and *ANY* Eiffel also allows to export a feature to specific classes, so called friends.

```
class B
feature {A}
        g is do σ0 end
end

class D inherit B end
```

In the above example only features defined in *A* and features of subclasses of *A* can call *g* in *B*. Java does also know the concept of friends: Members of a class with a default modifier[k] are only

[j] ISE's C compiler version 5.6 does accept the call but future versions of the compiler will behave the same way as the Java translation does. The Java behaviour is also consistent with the ECMA standard for Eiffel.

[k] Not public, protected or private.

visible to classes in the same package, these classes are friends of each other. But this concept is far away from the flexibility that Eiffel provides. In the example above *A* is a friend of *B* so they have to be in the same package, but what if *C* is a friend of *A*? Then **C** has to be in the same package as **A** and becomes a friend of **B** as well. Unfortunately a Java class can only be in one package[1]. Let us look at the clients of *B*

```
class A
feature
        f is do b.g end
        b: B is do create {D}Result end
end

class C
inherit A
feature
        h is do b.g end
end
```

And try to translate that:

```
public interface B {}
public class BImpl implements B {
    public void B_g() {σ0;}
}

public interface A {
    public void A_f();
    public B A_getB();
}

public class AImpl implements A {
    public void A_f() {f();}
    public B A_getB() {return getB();}

    protected void f() {bDelegate.B_g();}
    protected B getB() {return bDelegate;}
}

public interface C extends A {
    public void C_h();
    public B C_getB();
}

public class CImpl implements C {
    public void C_h() {h();}
    public void A_f() {f();}
    public B A_getB() {return getB();}
    public B C_getB() {return getB();}

    protected void f() {aDelegate.A_f();}
    protected void h() {((BImpl)getB()).B_g();}
```

[1]Of course usually that's a good thing, but not here.

```
32      protected B getB() {return aDelegate.A_getB();}
33  }
```

The interesting thing happens on line 31. To call B_g a **BImpl** is required but A_getB is of type **B**. Since A_getB does not return an object of type **BImpl** the cast will throw an exception. Not to make a cast at all will not work this time. The verifier will reject the program without a cast because there is no method B_g() defined in **B**. The only solution here is to handle features exported to friends as if they where exported to *ANY*:

Rule 14: Features exported to friends are handled the same way as features exported to *ANY*.

2.2.19 Expanded Types

Besides reference types Eiffel does also know the notion of value types. A variable or class defined as **expanded** is such a value type. Expanded types have two main properties:

1. A variable defined as expanded is never Void.

2. An expanded value is attached to its defining class and can't be shared among multiple objects.

The first property implies, that it is not required to create an expanded value, that it can be used right away without the need of instantiating an object. The second property implies that assignments are handled in a special way if expanded types are present. Since two different expanded variables can never reference to the same object it is also required to handle comparison differently, since comparing two references to expanded types for equality will always yield false. Let us look at an example first:

```
class VALUE_REF
feature
        item: INTEGER
        set_item(an_item: INTEGER) is
                do
                        item := an_item
                end
end

expanded class VALUE inherit VALUE_REF end

local
        v1: VALUE
        v2: VALUE_REF
do
        v1.set_item(100)

        create v2
        v2.set_item(100)

        v2 := v1
        if (v2 = v1) then σ0 end
end
```

There is nothing special about the translation of *VALUE_REF*. *VALUE* is translated as if it were a non expanded class. It's only important for clients to know that *VALUE* is expanded. The ECMA standard allows a compiler to handle an expanded class as if it were a reference type.[m] The only difference can be found in the client of such a class:

1. The client does not need to instantiate the class.
2. Value types have copy semantics which means that an assignment of a value type to another type will not result in assigning the reference to the value type but a reference to a clone of the value type.

Let's try to translate the client.

```
        Value v1 = new ValueImpl();
        v1.setItem(100);

        Value v2;
        v2 = ValueConstructor.defaultCreate();
        v2.setItem(100);

        v2 = v1.twin();
        if (equal(v1,v2)) {σ0;}
```

The interesting two cases are attachment of v1 to v2 and comparison of v1 with v2 for equality. Since v1 does have copy semantics a twin of the object referenced by v1 is attached to v2. Since two different variables can never reference the same value type a comparison of two references to value types is useless. The two values are instead compared with *equal*.

Rule 15: Assigning a value type to another type results in assigning a clone of the object referenced by the value type.

Rule 16: If a value type is involved in a comparison for equality then the objects and not the references are compared.

2.2.20 once

In Eiffel it is possible to mark a routine as **once**. Such routines are only executed once.

```
class SHARED
feature
        value: VALUE is
                once
                        create Result
                end
end
```

[m]ISE compiler 5.5 does not accept line 23 and 24, but it will in the future. We just make sure we are ready for that case. But the ECMA standard does not allow to assigning a reference type to a value type.

If *A* and *B* both inherit from *SHARED* then both *value* queries will return the same object:

```
1         local
2                 a: A
3                 b: B
4         do
5                 create a
6                 a.value.set_item (100)
7                 create b
8                 print(b.value.item.out)
9         end
```

The print statement on line 8 will print 100 to the console. *value* is executed (at most) once during the systems lifetime. This can be modelled using the singleton pattern:

```
public class SharedImpl implements Shared {

    public Value Shared_getValue() {return getValue();}

    protected Value getValue() {
        if (valueOnce == null) valueOnce = new ValueImpl();
        return valueOnce;
    }
    private static Value valueOnce;
}
```

There is also the possibility to write once procedures. A once procedure is only executed once during the systems lifetime. Any subsequent call after the first one has no effect. It's not much harder to model this behaviour in Java:

```
    private void doOnce() {
        if (!doOnceExecuted) {
            $\sigma_0$;
            doOnceExecuted = true;
        }
    }
    private boolean doOnceExecuted = false;
```

2.3 Basic types

Conceptually there are no types in Eiffel which require a special handling. An *INTEGER* for example is just an expanded *INTEGER_REF* and can be viewed as any other expanded type in the system. But there are two problems with this approach: 1. Not mapping *INTEGER* to its corresponding primitive type int in Java would make programs way to slow at execution time. 2. *INTEGER_REF* is defined by making extensive use of *INTEGER* and *INTEGER* is defined as expanded *INTEGER_REF* which leads to a bootstrapping problem. Therefore the so called Eiffel basic types are translated to Java primitive types:

Eiffel basic type	Java primitive type
BOOLEAN	int
INTEGER	int
DOUBLE	double
REAL	float
INTEGER_64	long
INTEGER_16	short
INTEGER_8	byte
CHARACTER	char
POINTER	int

For example the following function:

```
        add (x: DOUBLE; y: INTEGER): DOUBLE is
                do
                        Result := x + y
                end
```

is translated to:

```
Method protected  add (double,int) -> double
0       dconst_0
1       dstore  4
3       dload_1
4       iload_3
5       i2d
6       dadd
7       dstore  4
9       dload   4
11      dreturn
```

The logic that performs the special handling is already implemented by the CIL visitor. The only changes required was that in MSIL exists only one operator for every arithmetic function, whereas in Java there are multiple operators for every arithmetic function. Which operator has to be chosen depends on the types of the two top stack elements. For example dadd for a double addition but iadd for a integer addition. Therefore it was required to pass the type of the needed operation to the code generator:

```
        generate_binary_operator (code: INTEGER; type: TYPE_I) is
                        -- Generate a binary operator represented by 'code'.
                        -- Look in IL_CONST for 'code' definition.
                do
                        if is_integer (type) or else is_boolean (type) then
                                generate_integer_binary_operator (code)
                        elseif is_double (type) then
                                generate_double_binary_operator (code)
                        elseif is_real (type) then
                                generate_real_binary_operator (code)
                        elseif is_integer_64 (type) then
                                generate_integer_64_binary_operator (code)
                        end
                end
```

2.4 Code

So far there was no discussion about how to generate the code of feature bodies. The code is generated by applying a visitor to an AST. This is a well know technique and is discussed in Compilers: principles, techniques, and tools [1]. For example:

```
if $\sigma_0$ then
      $\sigma_1$
end
```

is translated to:

```
y          $\sigma_0$
x          ifne x+2
x+1        $\sigma_1$
x+2        ...
```

Translating other constructs like expressions, assignments, loops, conditions and so forth is straight forward and therefore not discussed in this report.

2.5 Contracts

There are six different kinds of assertions in Eiffel: Preconditions, postconditions, class invariants, loop variants, loop invariants and checks. All are implemented by using the same pattern. Each assertion is a set of boolean expressions. Each has to evaluate to true at runtime otherwise an Error is thrown and the execution stops at the position of the failed expression.

2.5.1 Checks

The simplest among the assertion is the check. Every boolean expression within the check block has to evaluate to true. For example the Eiffel program:

```
        local
            i: INTEGER
        do
            i := 100
            check
                i_is_100: i = 100
            end
        end
```

Is translated to Java like following:

```
1       int i = 100;
2       if (!(i = 100)) throw new Error("Check:␣i_is_100");
```

The execution will stop at line 2 if i is not equal to 100.

2.5.2 Preconditions

The pattern to handle preconditions is the same as the one for checks. Preconditions can be inherited from parent features if there are parent features. Since preconditions can only be weakened only one of the preconditions has to hold. For example:

```
class A
feature
        f (i: INTEGER) is
                require
                        i_positive: i >= 0
                do
                        σ0
                end
end

class B
inherit A redefine f end
feature
        f (i: INTEGER) is
                require else
                        i_in_range: i >= 0 and i <= 100
                do
                        σ1
                end
end
```

The added precondition to *f* in *B* is basically useless because the precondition for *f* in *B* is (i >= 0 and i <= 100) or (i >= 0) which is equal to i >= 0. But it's legal to write such preconditions in Eiffel. Let us look at the translation to Java:

```
class AImpl implements A {
        ...
        protected void f (int i) {
                if (!(i>=0)) throw new Error ("Precondition:␣i_positive");
                σ0
        }
}

class BImpl implements B {
        ...
        protected void f (int i) {
                if (!(i>=0 and i<=100))
                        if (!(i>=0)) throw new Error ("Precondition:␣i_positive");
                σ1
        }
}
```

First the precondition of *f* in *B* is evaluated, if the condition holds the method body is executed. If the condition does not hold then the preconditions of the parent features are executed until either one of the inherited preconditions hold or if non of them holds an error is thrown.

2.5.3 Postcondition

Postconditions are inherited from parent features as well. But unlike preconditions, postconditions can only be strengthened. Therefore all postconditions have to hold, the inherited ones as well as the new one. In addition one can also refer to the value of a variable as it was before the execution of the feature. This is expressed with the old keyword. The variable value is then stored in a temporary register at the beginning of the method.

```
class A
feature
        i: INTEGER

        f: INTEGER is
              do
                     σ0
              ensure
                     i_unchanged: i = old i
                     result_positive: Result >= 0
              end
end

class B
inherit A redefine f end
feature
        f: INTEGER is
             do
                    σ1
             ensure then
                    result_zero: Result = 0
             end
end
```

```
class AImpl implements A {
        ...
        protected int f() {
               int iold_ = getI();
               σ0
               if (!(i = iold_)) throw Error ("Postcondition:␣i_unchanged");
               if (!(result_>=0)) throw Error("Postcondition:␣result_positive");
               return result_;
        }
}

class BImpl implements B {
        ...
        protected int f() {
               int iold_ = getI();
               σ1
               if (!(result_ = 0)) throw Error ("Postcondition:␣result_zero");
               if (!(i = iold_)) throw Error ("Postcondition:␣i_unchanged");
               if (!(result_>=0)) throw Error("Postcondition:␣result_positive");
               return result_;
        }
}
```

2.5.4 Loop variants and invariants

In Eiffel it is possible to define loop variants and invariants. A loop invariant is a boolean expression that has to be true before the first execution of the loop and after every loop iteration. A variant is an integer expression, which is always non-negative and decreases on every iteration:

```
from
    i := 10
    j := 0
invariant
    i + j = 10
variant
    i
until
    i = 0
loop
    i := i − 1
    j := j + 1
end
```

This can be transformed to:

```
from
    i := 10
    j := 0
    check
        i + j = 10
        i >= 0
    end
    iold_ := i
until
    i = 0
loop
    i := i − 1
    j := j + 1
    check
        i + j = 10
        i >= 0 and i < iold_
    end
    iold_ := i
end
```

And this is exactly the way the loop invariants and variants are translated to Java:

```
int i = 10;
int j = 10;
if (!(i + j = 10)) throw new Error ("Loop␣invariant");
if (!(i >= 0)) throw new Error ("Loop␣variant:␣negative");
int iold_ = i;
while (!(i = 0)) {
    i = i − 1;
    j = j + 1;
    if (!(i + j = 10)) throw new Error ("Loop␣invariant");
    if (!(i >= 0)) throw new Error ("Loop␣variant:␣negative");
    if (!(i < iold_)) throw new Error ("Loop␣variant:␣non␣decreasing");
    iold_ = i;
}
```

2.5.5 Class invariants

The most challenging assertion type is the class invariant. Each class inherits all of the invariants of all the parent classes and each boolean expression has to hold before and after a feature is executed but only if it was a qualified call. A qualified call is a call that is not invoked on the current object. That's why it will not work to evaluate the invariant at the beginning and end of every method. Only the caller knows if he has to check the invariant. If it is a qualified call the caller has to check the invariants of the callee before and after the call, if it is not a qualified call then a check is not required. Therefore a method called invariant_() is added to the Java interface for *ANY*. All Java classes have to implement invariant_() therefore:

```
class A
feature
        i: INTEGER is 100
invariant
        i_positive: i >= 0
end

class B
feature
        j: INTEGER is -100
invariant
        j_negative: j < 0
end
```

```
class AImpl implements A {
        ...
        invariant_() {
             if (!(getI() >= 0)) throw new Error("Invariant:␣i_positive");
             anyDelegate.invariant_();
        }
}

class BImpl implements B {
        ...
        invariant_() {
             if (!(getJ() < 0)) throw new Error("Invariant:␣j_negative");
             aDelegate.invariant_();
        }
}
```

A qualified call to a method f() in **BImpl** will then look like following:

```
        B b = BConstructor.defaultCreate();
        b.invariant_();
        b.f();
        b.invariant_();
```

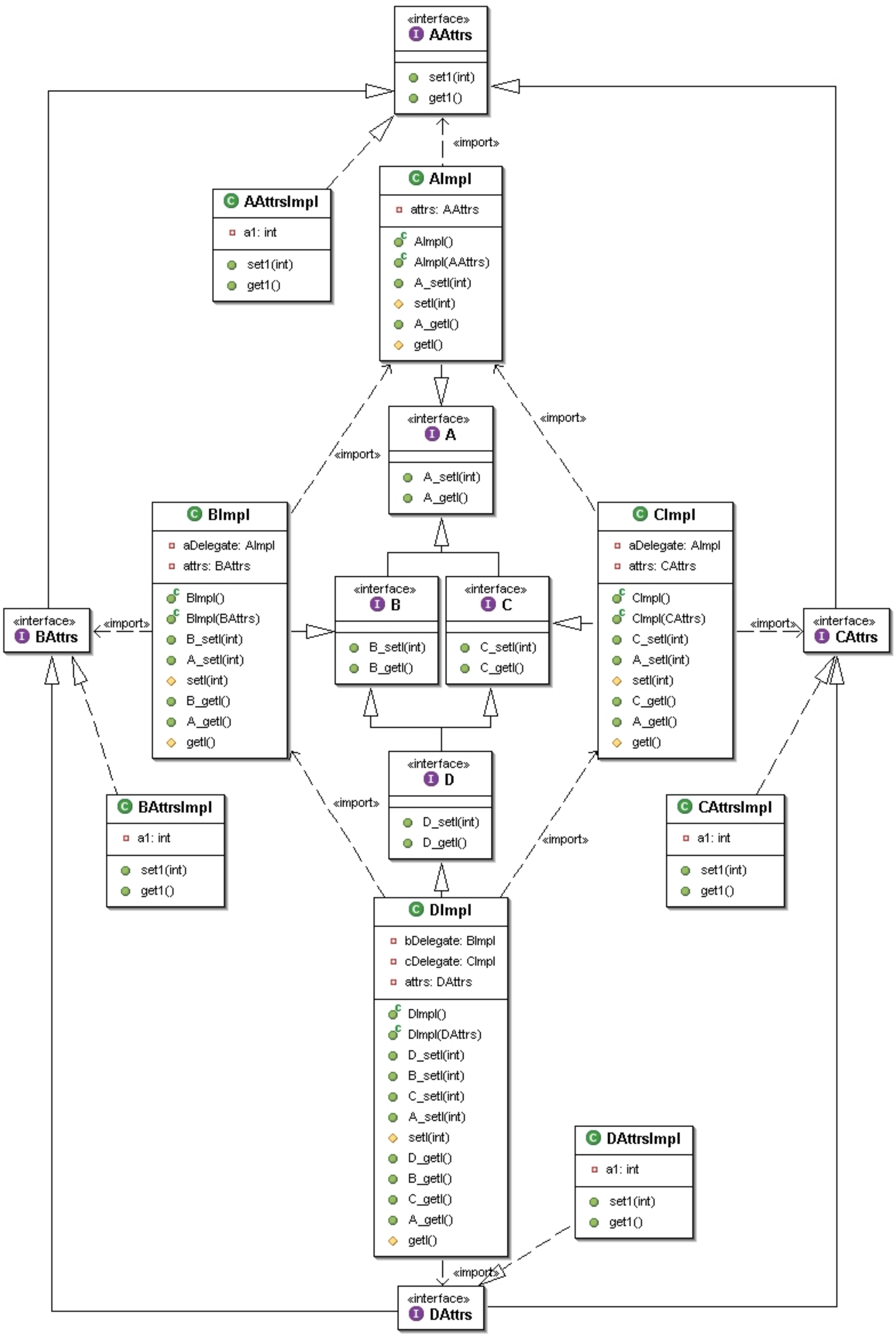

Figure 2.4: Translation with shared attributes.

Chapter 3

Implementation

3.1 Overview

The implementation of the Java Byte Code generator is discussed in this chapter.
The data flow chart on page 53 shows how Eiffel source code is translated to Java Byte Code. How ISE's compiler builds the AST and the *CLASS_C* instances is not described in this paper. How to build an AST from the source code of a class is a well known technique and is described for example in Compilers: Principles, techniques and tools [1]. It was not required to implement this for the Java Byte Code generator since this code is shared among all the code generators of the ISE compiler. The *CLASS_C* describes a compiled Eiffel class. For every compiled class in a system an instance of *CLASS_C* is created which holds the information for the compiled class. Among other things this includes all the features of the class, if the class is deferred, the name of the class or its parents.
The generator is divided into 3 main parts: First the *ISE_MAPPER* tries to find out with the help of *CLASS_C* and *FEATURE_I* that is built by the ISE compiler for every feature in the system from which parent class it came from. This information is needed to build the Java interfaces and class skeletons in a second step. In the last step the Eiffel code of the body of every redefined, effective or new feature is translated to Java Byte Code.

3.2 Mapper

In order to implement the code generator it is required to know for every feature f in a class A from which direct parent of A f was coming from. This information is stored in the data structure described by *FEATURE_INFO*:

```
class FEATURE_INFO
feature -- Access

        parent_features: LIST [FEATURE_INFO]
                        -- Features in direct parents of `associated_class'
                        -- which are ancestors of `Current' if any.
invariant
        parent_features_not_void: parent_features /= Void
        parent_features_not_contains_void: not parent_features.has (Void)
end
```

If f is new in A then *parent_features* is empty. The term new means that f was introduced in A and the origin of f is therefore A. Otherwise the parent features of f need to be known to delegate the call later on. The parent features of f are all the features in direct parent classes of A which are precursors of f. Due to sharing it is possible for a feature to have more than one parent feature. The information is collected in *ISE_MAPPER* for all classes in the system. The only purpose of *ISE_MAPPER* is to find out for every feature of a class where it was coming from. It was not possible to write a visitor for the AST to collect this information since it is not possible to write customised visitors for ISE's AST. Therefore it was required to collect this information from the data structures build by the ISE compiler for classes (*CLASS_C*) and features (*FEATURE_I*). The code of *ISE_MAPPER* is very specific and therefore not shown here. One would expect to have an Eiffel compiler which either has all the information which *FEATURE_INFO* holds present or is at least able to apply a customised visitor to the AST to collect the required information.

3.3 Name mangling

Implementing the name mangling is straight forward once the information about where a feature was coming from is present

```
f.add_mangled_name(create {MANGLING}.make(c.interface_name, f.java_name, f))
if not f.is_new and not f.is_unselected then
        from
                f.parent_features.start
        until
                f.parent_features.after
        loop
                f.add_mangle_table (f.parent_features.item.mangling_table)
                f.parent_features.forth
        end
end
```

This is done for all classes c in the system and for all features f in every class. To reduce memory consumption it is also possible not to do this explicitly but implicit while the classes and interfaces are generated. But for demonstration purposes this is done explicitly here. The class *MANGLING* has only three attributes: *interface_name*, *feature_name* and *associated_feature*. A list of manglings is added to *FEATURE_INFO* as well as commands to manipulate the list:

```
class interface FEATURE_INFO
feature -- Access

        mangling_table: LIST [MANGLING]
                        -- List with all the names for feature
feature -- Element change

        add_mangled_name (a_mangling: MANGLING) is
                        -- Add 'a_mangling' to 'mangling_table'.
                require
                        a_mangling_not_void: a_mangling /= Void
                        not_has: not mangling_table.has (a_mangling)
                ensure
                        has: mangling_table.has (a_mangling)

```

```
	add_mangle_table (a_table: LIST [MANGLING]) is
			-- Add all elements in 'a_table' to 'mangling_table'.
			-- makes sure every entry is unique.
		require
			a_table_not_void: a_table /= Void
		ensure
			added: a_table.for_all (agent mangling_table.has)

invariant
	mangling_table_not_void: mangling_table /= Void
end
```

Also of interest here is *interface_name* from *CLASS_TABLE* and *java_name* from *FEATURE_INFO*. These queries translate the Eiffel style name of a class respectively feature to Java style. Java uses camel case style, whereas Eiffel uses underscores to separate words.

```
	java_name: STRING is
			-- Java style name
		do
			if is_attribute and then is_getter_style then
				Result := "get" + camel_case (name, True)
			else
				Result := camel_case (name, False)
			end
		ensure
			result_not_empty: Result /= Void and then not Result.is_empty
		end

	interface_name: STRING is
			-- Java style class name for 'a_name'
		do
			Result := camel_case (name, True)
		ensure
			result_not_empty: Result /= Void and then not Result.is_empty
		end
```

The query *camel_case* is defined in *JAVA_STYLER*:

```
	camel_case (a_string: STRING; first_upper: BOOLEAN): STRING is
			-- 'a_string' in CaMeLcAsE with first character in upper case if 'first_upper'
			-- i.e. HASH_TABLE -> HashTable.
		require
			a_string_not_empty: a_string /= Void and then not a_string.is_empty
		local
			i, nb: INTEGER
			c: CHARACTER
			next_is_upper: BOOLEAN
		do
			if first_upper then
```

```
                        Result := a_string.item (1).upper.out
                    else
                        Result := a_string.item (1).lower.out
                    end
                    from
                        i := 2
                        nb := a_string.count
                        next_is_upper := False
                    until
                        i > nb
                    loop
                        c := a_string.item (i)
                        if c.is_equal ('_') then
                            next_is_upper := True
                        else
                            if next_is_upper then
                                Result.append_character (c.
                                    upper)
                                next_is_upper := False
                            else
                                Result.append_character (c.
                                    lower)
                            end
                        end
                        i := i + 1
                    end
                ensure
                    Result_not_void: Result /= Void and then not Result.
                        is_empty
                end
```

3.4 Generation

Two more preparation steps are required before the generation of code can start:

1. Every *CLASS_TABLE* object gets a reference to all its direct parents, *CLASS_TABLE* objects as well.

2. All parameters as well as return types of all features are referenced to the corresponding *CLASS_TABLE* instances.

The figure on page 54 shows part of the memory state before the code generation for the following small system starts:

```
class A
feature
    max (a: INTEGER; b: INTEGER): INTEGER is
        do
            if a > b then Result := a else Result := b end
        end
end
```

```
class B
inherit
        A rename max as maximum end
end
```

3.4.1 Generating interfaces

With all the preparations and with the help of the Java Byte code generation library [4] generating the interfaces is straight forward:

```
generate_interface (a_class: CLASS_TABLE) is
                        -- Generate a Java interface for 'a_class'.
        require
                a_class_not_void: a_class /= Void
        local
                f: FEATURE_INFO
                l_mangling: MANGLING
                l_parent: CLASS_TABLE
                class_gen: CLASS_GENERATOR
                method_generator: METHOD_GENERATOR
                l_file: JAVA_CLASS_FILE
        do
                create class_gen.make (a_class.interface_name)

                class_gen.set_access_flags (acc_public | acc_interface |
                    acc_abstract)
                class_gen.set_super_class ("java/lang/Object")
                class_gen.set_major_version (46)
                class_gen.set_minor_version (0)
                from
                        a_class.parents.start
                until
                        a_class.parents.after
                loop
                        l_parent := a_class.parents.item
                        class_gen.add_interface (l_parent.interface_name)
                        a_class.parents.forth
                end

                from
                        a_class.features.start
                until
                        a_class.features.after
                loop
                        f := a_class.features.item
                        if not f.is_exported_to_none then
                                l_mangling := f.mangling_table.first

                                create method_generator.
                                    make_from_constant_pool_gen (class_gen.
                                    constant_pool_gen)
                                method_generator.set_name (l_mangling.
                                    interface_name + "_" + l_mangling.
                                    java_name)
                                method_generator.set_access_flags (
                                    acc_public | acc_abstract)
                                method_generator.set_descriptor (f.
                                    descriptor)
                                class_gen.add_method (method_generator.
                                    method)
                        end

                        a_class.features.forth
                end

                create l_file.make_open_write (a_class.interface_name + ".
                    class")
                class_gen.java_class.dump_component (l_file)
                l_file.close
        end
```

Of interest may be the descriptor given to the method. A descriptor is a sequence of characters which describes the type of parameters and the type of the result of a method in Java.

```
        descriptor: STRING is
                        -- Java descriptor for 'Current'.
                do
                        from
                                Result := "("
                                parameters.start
                        until
                                parameters.after
                        loop
                                Result.append (parameters.item.descriptor)
                                parameters.forth
                        end
                        Result.append (")")
                        if has_result then
                                Result.append (type.descriptor)
                        else
                                Result.append ("V")
                        end
                ensure
                        result_not_void: Result /= Void
                end
```

The descriptor in *CLASS_TABLE* is shown below:

```
        descriptor: STRING is
                        -- Java descriptor
                do
                        if interface_name.is_equal ("Boolean") then
                                Result := "Z"
                        elseif interface_name.is_equal ("Integer") then
                                Result := "I"
                elseif interface_name.is_equal ("Double") then
                    Result := "D"
                elseif interface_name.is_equal ("Real") then
                    Result := "F"
                elseif interface_name.is_equal ("Integer64") then
                    Result := "J"
                elseif interface_name.is_equal ("Character") then
                    Result := "C"
                elseif interface_name.is_equal ("Integer8") then
                    Result := "B"
                elseif interface_name.is_equal ("Integer16") then
                    Result := "S"
                elseif interface_name.is_equal ("Pointer") then
                    Result := "I"
                else
                    Result := "L" + interface_name + ";"
                end
            ensure
                result_not_void: Result /= Void
            end
```

This reflects the mapping from Eiffel basic types to Java base types.

3.4.2 Generating classes

Generating the class which implements the interface requires more steps:

1. A Java class is generated with the name of the interface plus an Impl postfix. If the Eiffel class is deferred then the Java class is abstract and another class with the name of the interface plus an ImplConcrete postfix is generated. This class is not abstract.

2. For every parent a field is generated which will hold a reference to the delegation object.

3. A Java class is generated with the name of the interface plus an AttrsImpl postfix. This class implements the corresponding Attrs interface.

4. A constructor with a parameter of the type of the Attrs interface is created in which all the delegation objects are instantiated by passing the parameter to there constructor and assigned the generated object to the delegation fields.

5. A default constructor is created which calls the constructor generated in the previous step.

6. Then for every feature f

   (a) If f is deferred then a protected abstract method is added to the Impl class and a concrete method with an empty body to ImplConcrete.

   (b) Otherwise if f is new, effected or redefined then the visitor on the AST to start generating the method body code is invoked.

   (c) Otherwise the implementation of one of the parent features of f is called through one of the delegation objects.

   (d) In a last step all the manglings of f are generated which call the implementation of f generated in the previous step.

7. If the class is the root class then a public static void main is generated in the Java class which calls the creation method make of the root class.

3.4.3 Generating code

The same visitor as the one for the CIL generator is used for the JBC generation. The visitor itself does not generate any code but calls commands on the deferred class *IL_CODE_GENERATOR*. This class is effected by *JAVA_CODE_GENERATOR* and an instance is passed to the visitor. The code is then emitted in *JAVA_CODE_GENERATOR*. A small section of this class is shown below. The two commands push default values of different types on top of the execution stack.

```
	put_default_value (type: TYPE_I) is
			-- Put default value of 'type' on stack.
		do
			if
				is_boolean (type) or else
				is_integer (type)
			then
				method_body.extend (create {INSTRUCTION}.
				    make (iconst_0))
			elseif is_double (type) then
				method_body.extend (create {INSTRUCTION}.
				    make (dconst_0))
			elseif is_real (type) then
				method_body.extend (create {INSTRUCTION}.
				    make (fconst_0))
			elseif is_integer_64 (type) then
				method_body.extend (create {INSTRUCTION}.
				    make (lconst_0))
			else
				check
					not_is_basic: not type.is_basic
				end
				put_void
			end
		end

	put_void is
			-- Add a Void element on stack.
		do
			method_body.extend (create {INSTRUCTION}.make (
			    aconst_null))
		end
```

Implementing most of these commands is straight forward.

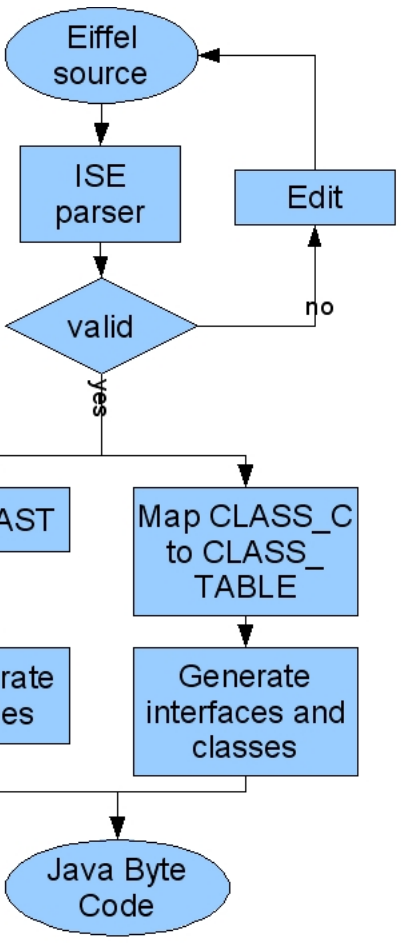

Figure 3.1: Flow chart for the Java Byte Code generator.

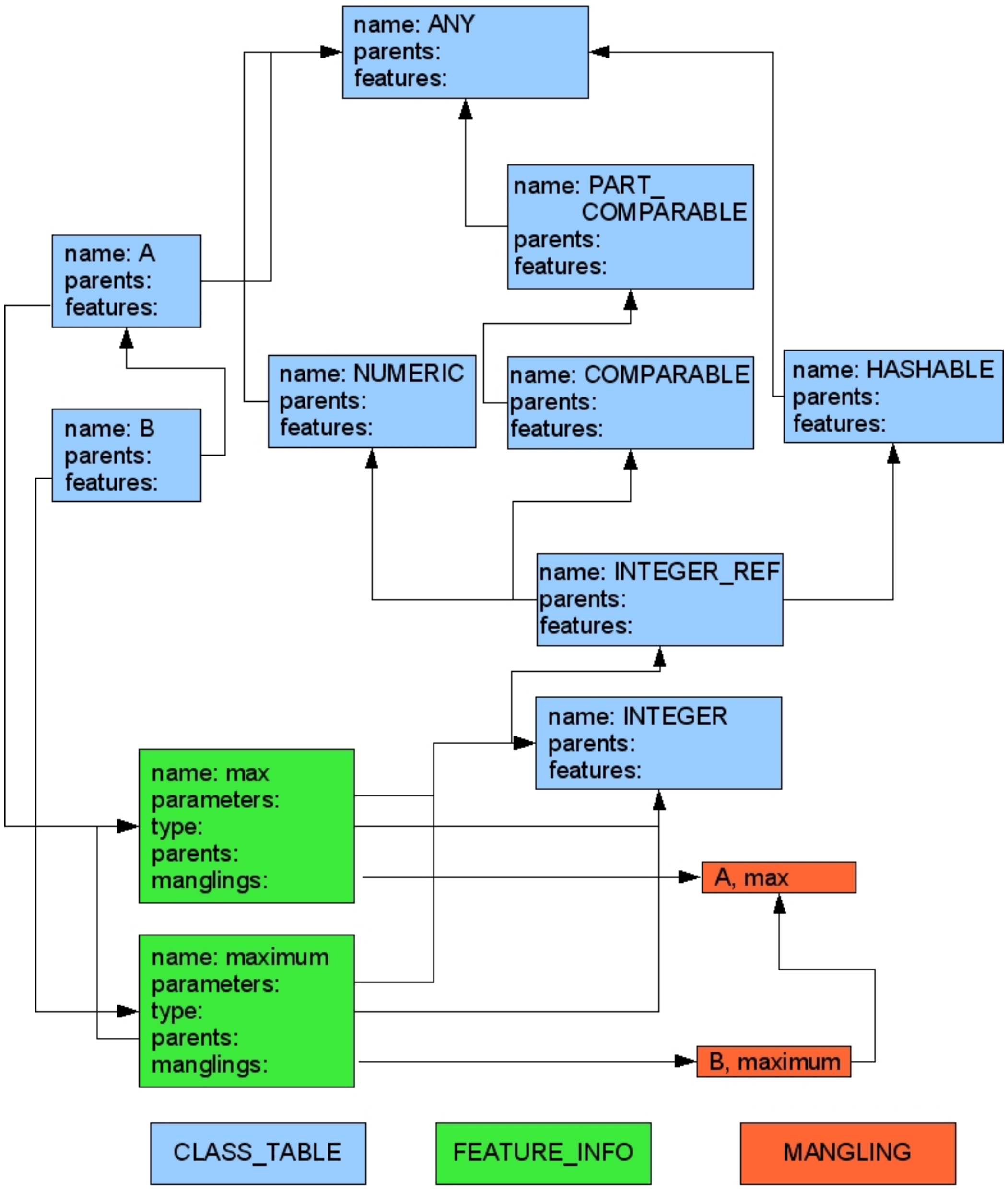

Figure 3.2: Part of the memory before code generation starts for system at page 48.

Chapter 4

Missing parts

Unfortunately it was not possible to implement all of the Eiffel constructs within the 4 months time frame given to write this thesis. The table below gives an overview about all the Eiffel constructs and whether they are implemented in the Java Byte Code generator and if not if there is at least the concept described in chapter 2 how to translate these constructs to Eiffel.

Construct	Concept	Implementation
Single class	X	X
Single inheritance	X	X
Multiple inheritance	X	X
Inherit multiple times from same class	X	X
Attributes	X	X
Constructors	X	-
Renaming	X	X
Redefining	X	X
Precursors	X	X
Sharing	X	X
Join with select	X	X
Deferred	X	X
Undefine	X	X
Covariance	X	X
Export	X	X
Descendent hiding	X	-
Expanded types	X	-
onces	X	-
Genericity	-	-
Agents	-	-
Contracts	X	X
Basic types	X	X
Arrays	-	-
Code	X	X

Construct	Concept	Implementation
Constants	X	X
Unique	-	-
Cloning and copying	-	-
Frozen	-	-
Rescue	-	-
Tuples	-	-
Infixes and prefixes	-	-
Eiffel base	-	-

Chapter 5

Case study

5.1 Overview

In this chapter a small program is introduced which is partly written in Eiffel and partly written in Java. The example application is TicTacToe a game where two opponents try to occupy 3 fields in a row. A row can be horizontal, vertical or diagonal. The game ends if either one of the players has occupied 3 fields or no more fields are free. In the presented TicTacToe version one opponent will be the computer the other a human. The data structure as well as the artificial intelligence is written completely in Eiffel and compiled to Java Byte Code. The view is written in Java and is basically a JFrame. The example will show how to integrate the generated Java Code into a Java program.

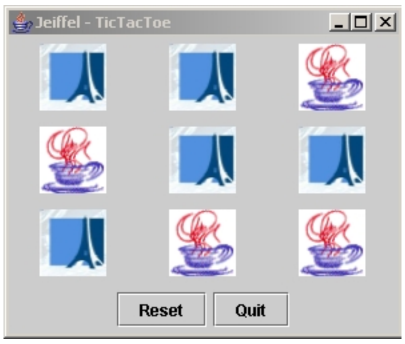

Figure 5.1: Screen shot of TicTacToe

5.2 Design

The main part of the program is written in Eiffel: The data structure *BOARD* as well as the artificial intelligence which can calculate moves for the computer player (*AI*, *AI_LAZY* and *AI_CLEVER*). The view **TicTacToeFrame** is written in Java and is a subclass of **JFrame**. To inform a view about state changes in the data structure a *BOARD* holds a reference to a *BOARD_CALLBACK* object. Whenever a field is set on the board the feature *set* on *BOARD_CALLBACK* is called.

BOARD_CALLBACK is deferred. The Java class **BoardCallbackPanel** inherits from *BOARD_CALLBACK* respectively from its translated version, **BoardCallbackImpl**, and implements the abstract protected method set(). Normally an Eiffel programmer would not need to write the class *BOARD_CALLBACK* because an agent passed to *BOARD* could do the same, but since JEiffel does not support agents yet a deferred callback class is added to the system. This may also be more natural for Java programmers since they are not used to agents.
In the game one player will be the user of the game and the other player will be the computer. Therefore some kind of artificial intelligence is needed. The base class for all artificial intelligence algorithms is the deferred class *AI* which has two subclasses: *AI_LAZY* and *AI_CLEVER*. To make the example a bit more interesting *AI_CLEVER* inherits from *AI* and from *AI_LAZY*.
The Figure below shows a high level overview of the Design. For clarity all classes implemented in Eiffel are shown as BON figures and all classes implemented in Java are shown as UML figures.

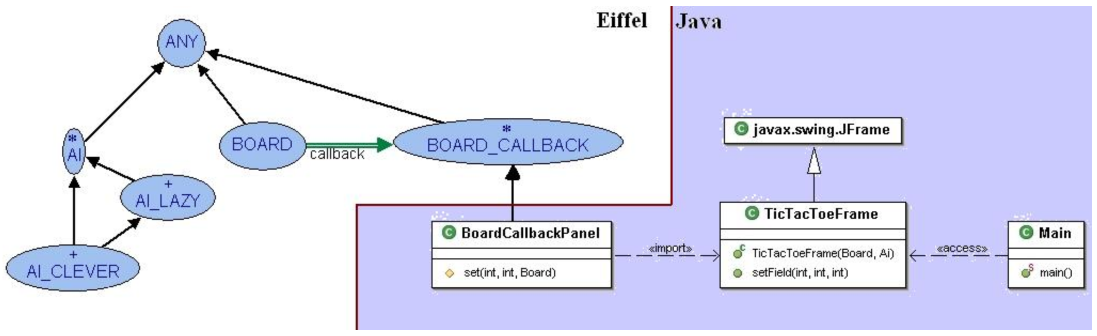

Figure 5.2: Design for TicTacToe

5.3 Implementation

5.3.1 Eiffel side

The core of the program is its data structure: *BOARD*. The interface view for it is shown next:

```
class interface BOARD
create make
feature -- Initialization

        make (a_callback: BOARD_CALLBACK)
                        -- Create the board
                require
                        a_callback_not_void: a_callback /= Void
                ensure
                        set: callback = a_callback
feature -- Access

        number_of_columns: INTEGER is 3
                        -- Number of columns on the board

        number_of_rows: INTEGER is 3
                        -- Number of rows on the board
```

```
feature -- Status report

        has_free_field: BOOLEAN
                        -- Does at least one free field exist?

        has_winner: BOOLEAN
                        -- Does a player win?
                ensure
                        defined: Result = is_computer_winner or
                            is_human_winner

        is_computer (column: INTEGER; row: INTEGER): BOOLEAN
                        -- Is field at position ('column', 'row') taken by
                            the computer?
                require
                        row_in_range: row > 0 and row <= number_of_rows
                        column_in_range: column > 0 and column <=
                            number_of_columns

        is_computer_winner: BOOLEAN
                        -- Does computer own 3 fields in a row?

        is_free (column: INTEGER; row: INTEGER): BOOLEAN
                        -- Is field at position ('column', 'row') not taken
                            by any player?
                require
                        row_in_range: row > 0 and row <= number_of_rows
                        column_in_range: column > 0 and column <=
                            number_of_columns
                ensure
                        defined: Result = (not is_computer (column, row) and
                             not is_human (column, row))

        is_human (column: INTEGER; row: INTEGER): BOOLEAN
                        -- Is field at position ('column', 'row') taken by
                            the human?
                require
                        row_in_range: row > 0 and row <= number_of_rows
                        column_in_range: column > 0 and column <=
                            number_of_columns

        is_human_winner: BOOLEAN
                        -- Does human own 3 fields in a row?

feature -- Element change

        reset
                        -- Set all fields to be non occupied.



        set_field (column: INTEGER; row: INTEGER; is_computer_taken: BOOLEAN)
                        -- Set field at position (`column', `row') to
                           computer owned if `is_computer_taken'
                        -- to human owned otherwise.
                require
                        row_in_range: row > 0 and row <= number_of_rows
                        column_in_range: column > 0 and column <=
                           number_of_columns
                ensure
                        set: (is_computer_taken and is_computer (column, row
                           )) or is_human (column, row)

invariant
        number_of_rows_greater_zero: number_of_rows > 0
        number_of_columns_greater_zero: number_of_columns > 0
        callback_not_void: callback /= Void
end
```

BOARD stores the state of all the 9 fields on the board. A field is either free, occupied by the computer or occupied by the human player. *BOARD* also provides queries to find out if a player has won the game and of course a command to occupy a field and to free all the fields on the board. Since JEiffel does not support arrays it was required to add 9 *INTEGER* attributes, for each field one, to *BOARD*. This makes the implementation a bit ugly. But this implementing detail is hidden and a client has therefore not to worry about it.
To connect the Java view with the data structure the deferred class *BOARD_CALLBACK* is added to the system:

```
deferred class BOARD_CALLBACK
feature -- Callback

        set (column: INTEGER; row: INTEGER; board: BOARD) is
                        -- Field on `board' at position (`column', `row')
                           has changed.
                require
                        board_not_void: board /= Void
                deferred
                end

end
```

BOARD calls *set* on its *BOARD_CALLBACK* object whenever a field on the board is set. Another class can inherit from *BOARD_CALLBACK* and effect *set*. A view can pass an instance of the effective class to *BOARD* to get informed about state changes in the data structure and to update the view accordingly.
The most interesting class is *AI_CLEVER* which inherits from *AI* directly and indirectly through *AI_LAZY*. The class *AI* is shown next:

```
deferred class AI
feature -- Access

        last_row: INTEGER
                        -- The row calculated by 'calculate_move'

        last_column: INTEGER
                        -- The column calculated by 'calculate_move'

feature -- Operations

        calculate_move (a_board: BOARD) is
                        -- Calculate move on 'a_board'. Store result
                        -- in 'last_column' and 'last_row'.
                require
                        a_board_not_void: a_board /= Void
                        has_free_field: a_board.has_free_field
                deferred
                ensure
                        found_valid_row: last_row > 0 and last_row <=
                            a_board.number_of_rows
                        found_valid_column: last_column > 0 and last_column
                            <= a_board.number_of_columns
                        found_free_field: a_board.is_free (last_column,
                            last_row)
                end
end
```

The class *AI_LAZY* inherits from *AI* and effects *calculate_move*. The implementation of *calculate_move* in *AI_LAZY* is a very simple one: It just iterates through the fields on the board and stores the position of the first free one in *last_row* and *last_column*. The interesting class is *AI_CLEVER* which contains a more suffisticated implementation for *calculate_move*:

```
class AI_CLEVER
inherit
        AI select calculate_move end
        AI_LAZY rename calculate_move as calculate_lazy_move end

feature -- Operation

        calculate_move (a_board: BOARD) is
                        -- Calculate move on 'a_board'. Store result
                        -- in 'last_column' and 'last_row'.
                do
                        if not prevent_winning (a_board) then
                                calculate_lazy_move (a_board)
                        end
                end




```

```
feature {NONE} -- Implementation

        prevent_winning (a_board: BOARD): BOOLEAN is
                        -- Make a move to prevent winning of opponent.
                        -- If no such move is required return False.
                require
                        a_board_not_void: a_board /= Void
                do
                        ...
                end
end
```

AI_CLEVER renames the *calculate_move* from *AI_LAZY* to *calculate_lazy_move* and selects *calculate_move* from *AI*. In *calculate_move* either a move is made to prevent the opponent from winning or if no such move is required *calculate_lazy_move* is called which just occupies the first free field. From a design point of view it would be of course better if *AI_CLEVER* just inherits from *AI* and uses *AI_LAZY* as a client, since it does not make much sense for *AI_CLEVER* to be also an *AI_LAZY*. The design above was only chosen because it is the much more interesting case: The two attributes *last_row* and *last_column* are shared among *AI_LAZY* and *AI_CLEVER*. Both translated Java classes have to operate on the same attributes. How this is achieved is explained in section 2.2.10. The implementation of *prevent_winning* is not of great interest here. It uses numerous control sequences like loops and if then elses as well as arithmetic operations on integers like additions and modulo. It checks if it is possible for the opponent to occupy 3 fields in a row in the next step and if so the position of the free field in the row is stored in *last_row* and *last_column* otherwise False is returned.

5.3.2 Java side

On the Java side three classes have been written: A class **Main** which contains the main method and sets up the system, a class **TicTacToeFrame** which is the view for a *BOARD* and on which the user can select fields to occupy and a class **BoardCallbackPanel** which is a *BOARD_CALLBACK* and which informs a **TicTacToeFrame** of state changes in *BOARD*. All of these classes are very simple. Of interest is how to use the translated Eiffel classes in Java. Let us look at **Main** first:

```
public class Main {

        public static void main(String[] args) {
                BoardCallbackPanel boardCallback = new BoardCallbackPanel();

                Board board = new BoardImpl();
                board.Board_make(boardCallback);

                Ai ai = new AiCleverImpl();

                TicTacToeFrame frame = new TicTacToeFrame(board, ai);
                boardCallback.setView (frame);

                frame.setTitle("Jeiffel␣-␣TicTacToe");
                frame.show();
        }
}
```

First an instance of **BoardCallbackPanel** is created which is then passed to a new instance of **BoardImpl**. The instance of **BoardImpl** as well as an instance of **AiCleverImpl** is passed to a new **TicTacToeFrame** which is then shown at the end of the main method.
BoardCallbackPanel is pretty simple. It inherits from the abstract class **BoardImpl** and implements the protected method set in which the view is informed about any state changes:

```java
public class BoardCallbackPanel extends BoardCallbackImpl {
        private TicTacToeFrame frame;

        public void setView (TicTacToeFrame frame) {
                this.frame = frame;
        }

        protected void set (int column, int row, Board board) {
                if (board.Board_isComputer(column, row)) {
                    frame.setField (row - 1, column - 1, 2);
                } else if (board.Board_isHuman(column, row)) {
                    frame.setField (row - 1, column - 1, 1);
                } else {
                    frame.setField (row - 1, column - 1, 0);
                }
        }

}
```

A nicer design would be that **TicTacToeFrame** inherits from **BoardCallbackImpl**. But **TicTacToeFrame** already inherits from **JFrame** and since Java does not support multiple inheritance the class **BoardCallbackPanel** is required as adapter.
The **TicTacToeFrame** is mostly generated with a gui builder[a]:

```java
public class TicTacToeFrame extends javax.swing.JFrame {

        public TicTacToeFrame(Board board, Ai ai) {
                super();
                this.board = board;
                this.ai = ai;
                initGUI();
        }

        public void setField(int i, int j, int value) {
            ... Display right image at (i,j)
        }

        private void fieldSelected(int i, int j) {
                if (board.Board_isFree(i, j)) {
                        board.Board_setField(i, j, false);

                        if (board.Board_isHumanWinner()) {
                                putMessage("You␣are␣the␣winner!");
                                reset();
                        } else if (board.Board_hasFreeField()) {
```

[a]http://cloudgarden.com/jigloo/

```
                                ai.Ai_calculateMove(board);
                                board.Board_setField(ai.Ai_getLastColumn(),
                                    ai.Ai_getLastRow(), true);
                                if (board.Board_isComputerWinner()) {
                                        putMessage("The␣computer␣wins!");
                                        reset();
                                }
                        } else {
                                putMessage("no␣one␣wins!");
                                reset();
                        }
                }
        }

        private void reset() {
                board.Board_reset();
        }

        private void putMessage(String s) {
                JOptionPane.showMessageDialog(this, s);
        }

        private void initGUI() {
                ... Auto generated code
        }

        private Board board;
        private Ai ai;
}
```

Only the method fieldSelected is of greater interest. It is called whenever the user clicks on a field. The position of the selected field is passed as parameter. The method shows how to use a **Board** as a client.

5.4 Translation

The figures on page 65 and on page 66 show the UML diagram of all the generated classes and there relations.
The figure on page 65 shows the generated interfaces and implementing classes. One can see that **BoardCallbackImpl** is extended by **BoardCallbackImplConcrete** which implements set. Why **BoardCallbackImplConcrete** is required is explained in section 2.2.14. Also of interest is the interface **AiClever** which inherits twice from **Ai**, once directly and once through **AiClever**. This reflects the exact same subtyping hierarchy as in the Eiffel code. Also hint that the method calculateLazyMove is added to **AiClever** since calculateMove from **AiLazy** was renamed to calculateLazyMove in **AiClever** and the implementation of calculateLazyMove will therefore call calculateMove on an instance of **AiLazyImpl**.
The figure on page 66 shows the generated interfaces and implementing classes for all the attributes.

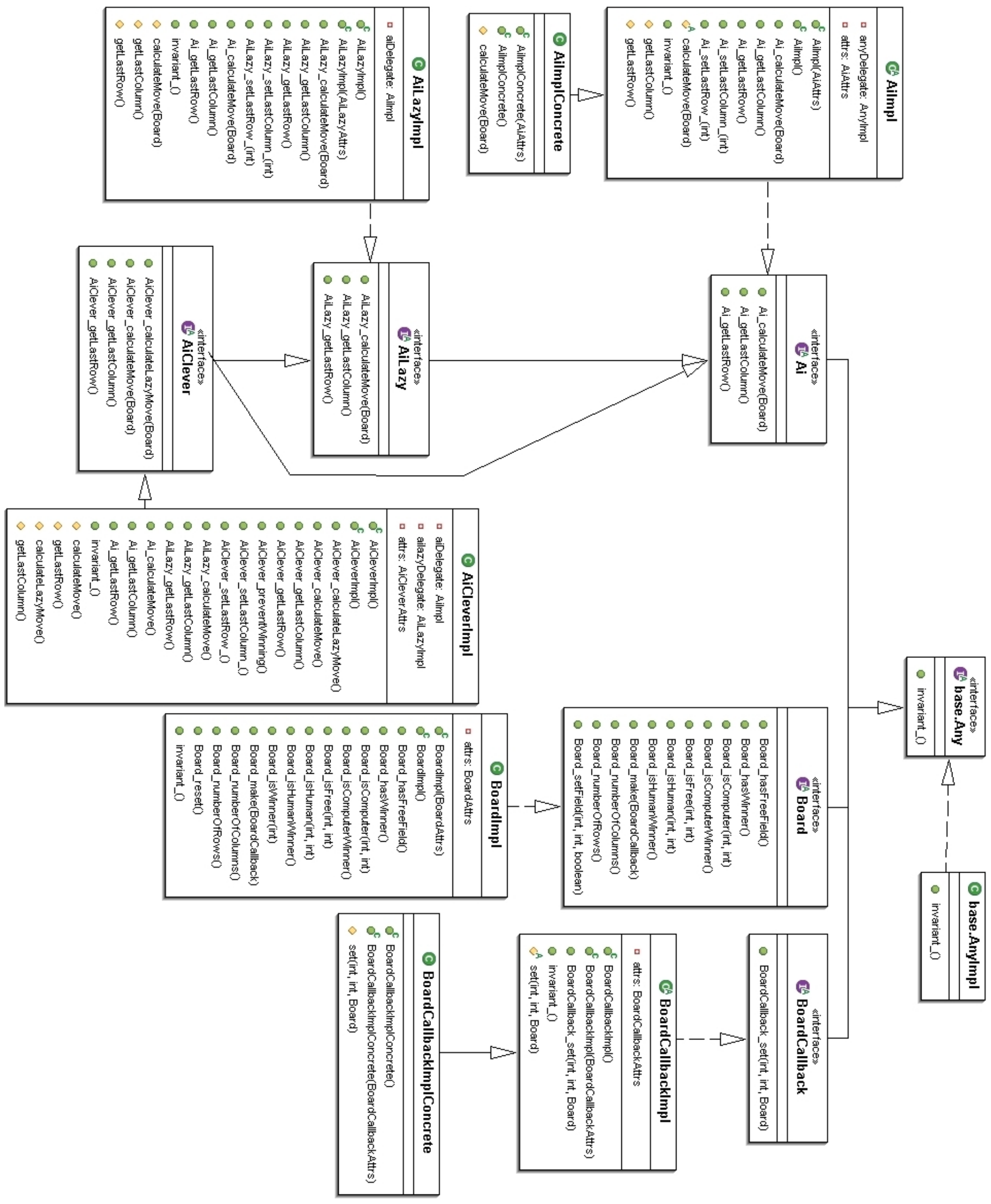

Figure 5.3: Class diagram of translated code

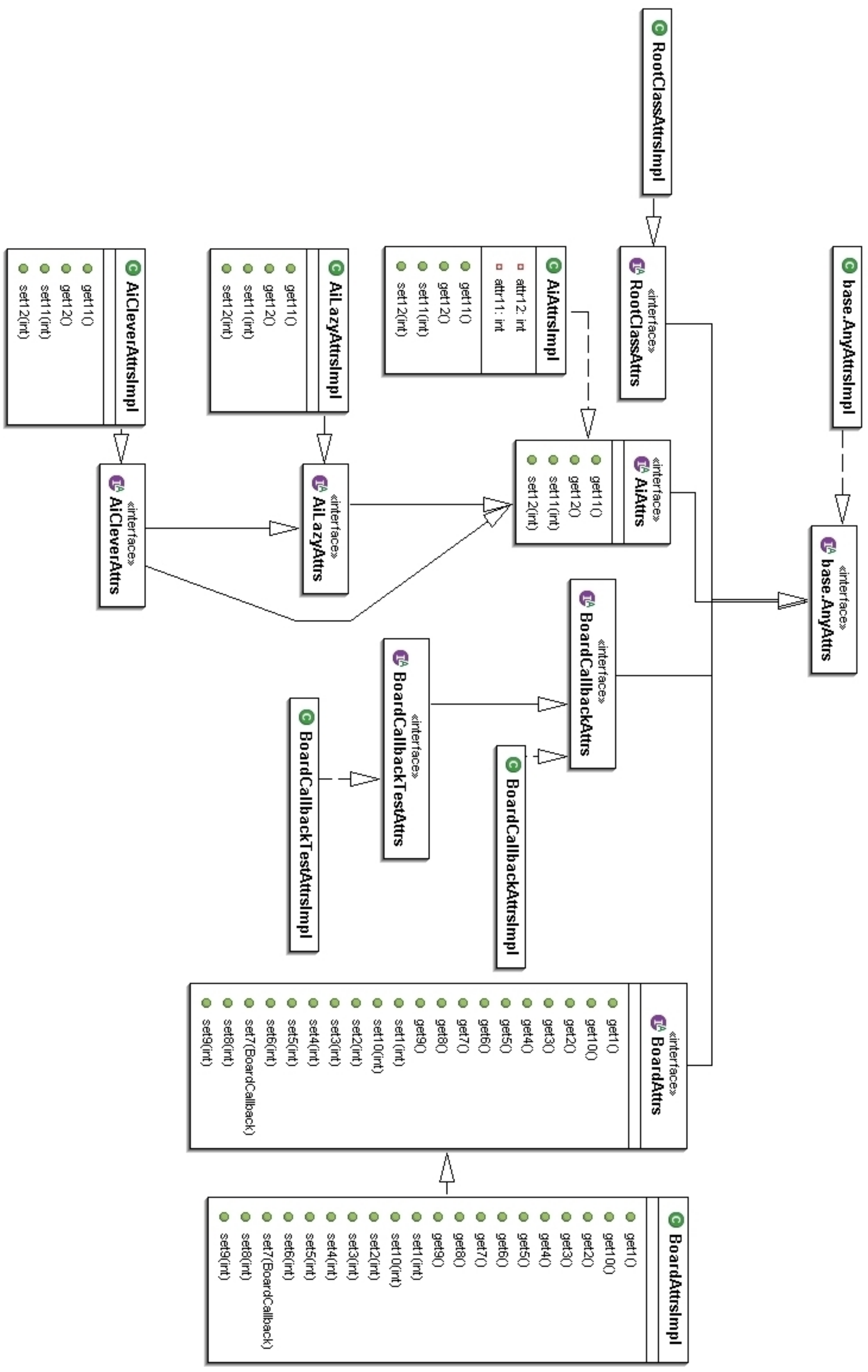

Figure 5.4: Class diagram of translated code for attributes

Chapter 6

Conclusion

6.1 Overview

It has been shown how to map a substantial subset of Eiffel to Java and how to implement a Java Byte Code generator backend for the ISE compiler.

6.2 From Eiffel to Java

To map Eiffel to Java one has to model multiple inheritance in Java. This is achieved by generating a Java interface for every Eiffel class in the system. The interfaces have subtyping relations among each other, the same relations as the corresponding Eiffel classes. For every interface a class is generated which implements this interface. This class contains the Java Byte Code which was generated from the source code by applying a visitor to the AST of every new, effected or redefined feature in the Eiffel class. If a feature in the Eiffel class is neither new, effected nor redefined then the implementation as defined in a parent class is executed. This is achieved by delegating calls to such methods to instances of other implementing classes. Each implementing class holds zero to many references to other implementing classes: One reference for every class which was a parent class in Eiffel.

The main difficulty in mapping Eiffel to Java is Eiffel's renaming facility. Renaming allows to assign a feature of a parent class another name in a subclass. In some situations, one exemplary shown in section 2.2.4, the renaming construct has to lead to a binding algorithm at runtime which is more complex then dynamic binding as implemented in the JVM. In this situations not only the type of an object at runtime has to be taken into consideration when a binding to an implementation is done, but also the type of the expression returning the reference to the object on which the call is invoked. In Java it is not possible to model this behaviour, because in Java only the runtime type of an object matters when a binding to an implementation is made by the JVM[a]. The chosen solution among multiple possible solutions presented in section 2.2.4 is name mangling. Name mangling is presented in section 2.2.5. To distinguish two different features with the same signatures two different names are given to the translations of this features.

Renaming is the main problem but there are many other problems that need some thought. All of

[a]If the method to invoke is a non static public one

them are discussed in chapter 2. An interesting case worth mentioning here is covariance. Covariance may lead to non type safe programs in Eiffel. Although Java is proven to be type safe (Java is type-safe [7]) this seams not to be the case for the Java Byte Code verifier. It is possible to execute a non type safe program on the JVM. The program is not rejected by the verifier prior to execution and the program does not crash at runtime. It is therefore possible to translate Eiffel's covariance to Java Byte Code without the requirement of any checkcasts at runtime or any other measures.

6.3 Implementation

To implement the Java Byte Code generator an additional backend was developed for the ISE compiler. This backend shares substantial parts of code with the common intermediate language generator, another backend for the ISE compiler. For example both generators share the same visitor for the AST. The generation is divided into two main steps:

1. Java interfaces and class skeletons are generated for every class in the system.
2. The empty method bodies of every class is either filled with the Java Byte Code generated from the Eiffel source, or a call to another implementing class is generated.

The most difficult part in implementing the Java Byte Code generator was to find out from which parent class a feature was inherited. Writing the actual code generator was then straight forward for most parts. Due to the limited time frame given to write this thesis it was not possible to implement all of the Eiffel language constructs and to translate Eiffel base. The main missing parts at the moment are: Genericity, agents and arrays. A detailed overview about the missing parts can be found in chapter 4.

6.4 Outlook

It is uncertain at the moment if a Java Byte Code generator as proposed in this report will every be part of an Eiffel Studio release. It is possible to translate Eiffel to Java but the generated code is, due to name mangling, not easy to understand and to use. Before it would be possible to include the code generator to Eiffel Studio, the remaining parts need to be implemented and Eiffel base needs to be translated to Java. It's not required to translate all of the classes of Eiffel base to Java, but some need a manual translation because some Eiffel base classes highly depend on the C runtime which is not present in Java.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 39, 45

[2] Dominique Colnet and Olivier Zendra. Targeting the Java Virtual Machine with Genericity, Multiple Inheritance, Assertions and Expanded Types. Research Report A00-R-137, LORIA - UMR 7503, September 2000. 1

[3] David Flanagan. *Java in a nutshell (2nd ed.): a desktop quick reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. 4

[4] Daniel Gisel. Eiffel library to generate java bytecodes, May 2003. 4, 49

[5] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. 1, 4

[6] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition edition, March 2000. 4, 29, 32

[7] Tobias Nipkow and David von Oheimb. Javalight is type-safe definitely. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170, New York, NY, USA, 1998. ACM Press. 68