From Eiffel to the Java Virtual Machine

Adding a new language target to the EiffelStudio compiler

Søren Engel, soen@itu.dk

Supervisor Dr. Joseph R. Kiniry

Thesis submitted to The IT University of Copenhagen
for the degree of Bachelor of Science in
Software Development

August 2012

Abstract

Previous attemps has been made towards translating and executing programs written in Eiffel on the Java platform, as described in Baumgartner [1]. However, the generated Java code has not been easy to use in terms of interoperability between the two languages, due to the restrictions of the Java language.

Due to the evolution of the Java language, the work presented in this thesis examines whether it is possible to simplify the translation model of Eiffel to Java. In doing so, this thesis describes a refined translation model that leverages on some of the new features of the Java language, such as the `invokedynamic` bytecode instruction. Moreover, in order to verify the correctness of the proposed translation model, a description follows on how the translation model may be integrated into the existing EiffelStudio compiler, in terms extending the back-end to target the Java platform.

In regards of simplicity, it was found that by using new language features of Java, the translation model of Eiffel to Java could be simplified to the extend that it solves the known issues found in the solution presented in Baumgartner [1]. In trying to integrate the translation model into the existing EiffelStudio compiler, it was found that no public documentation exists which described the internal structure of the EiffelStudio compiler. Neither did a library exists, by which it was possible to generate Java `.class` files. Consequently, this thesis presents a thorough description of both the infrastructure of compiler, as well as a general purpose library written in Eiffel that makes it possible to generate `.class` files.

Preface

Who should read this thesis

The content presented in this thesis is for readers with a background in computer science, preferably advanced undergraduate students. In order to gain most from this report, one should be fairly comfortable with the following subjects:

- Object-oriented programming languages (knowing C# or Java will be of great help)
- Basic compiler design and construction
- Software architectures and design patterns

However, it is not required that the reader knows the Eiffel programming language, nor is it a requirement to have worked with the EiffelStudio IDE or EiffelStudio compiler, in order to understand the material presented in this thesis. Where necessary, the relevant concepts will be explained as they are presented.

How to read this thesis

This thesis is organized into these listed six parts:

- *Introductory:* The first part describes the motivation behind this project as well as the goals. Chapter 2 gives a description of the Java Virtual Machine (JVM), the Eiffel programming language as well as revisiting some of the concepts of compiler design.
- *Analysis:* The next part contains the analysis in which a discussion on how to go from Eiffel to JVM bytecode is included, this is followed by a description of how the various constructs in Eiffel can be mapped onto the JVM object code. At the end of

chapter 3 is also included a section, which contains a technical explaination of how the compilation process in EiffelStudio works, from both at a high-level and low-level view.

- *Implementation:* This part describes the implementation that has emerged from the analysis, which includes a description of a library that can be used to generate JVM bytecode, as well as the work of extending the EiffelStudio compiler with a new language target generator.
- *Evaluation and Verification:* This section shortly discuss how the translation from Eiffel to JVM bytecode may be verified.
- *Conclusion:* Lastly the final part includes a reflection/conclusion on the work of this project, as well as looking at the future work it may be subject to.

Acknowledgements

I would like to start by recognizing my supervisor, Dr. Joseph R. Kiniry. Through this project he has been my mentor as well as catalyst during many sessions of brainstorming. I would also like to thank Emmanuel Stapf, Chief Engineer at Eiffel Software, for all his invaluable efforts in assisting and supporting me in doing this project. If it had not been for his calmness and "ease of working with" qualities, this project would not have reached the same result. Benno Baumgartner and David Raymond Christiansen also deserves a thank for both supporting me.

Next I would like to thank Thomas Hallier Didriksen and Sune Alkærsig, whom I have had the pleasure of sharing many thoughts and ideas with, while getting the grasp on both the Eiffel programming language, but also in working my way down in the source code of EiffelStudio.

Last, I would like to thank Niklas Schalck Johansson, who deserves special attention for his thorough review of this thesis.

Contents

Chapter 1

Background

1.1 Eiffel - The language

Eiffel is a standardized (International [2]) object-oriented language designed by *Bertrand Meyer* and the company *Eiffel Software*. The overall design goal behind the Eiffel language, its libraries and programming methodologies, is to enable programmers to create reliable, reusable software modules.

The design of Eiffel is based on a pure object-oriented programming theory[1], only minor influenced by other paradigms and concerns for supporting legacy code.

As a programming language Eiffel supports multiple inheritance, genericity, polymorphism, encapsulation, type-safe conversions and parameter covariance. However, Eiffel's most important contribution to software engineering is the concept called Design By Contract™ (DbC). DbC allows programmers to use assertions, preconditions, postconditions and class invariants to help them ensure program correctness without sacrificing efficiency, as well as to express and embed design decisions into the software they create.

Today, many of the concepts that were initially introduced by Eiffel has later found their way into Java, C#, and other contemporary languages.

1.1.1 EiffelStudio - The Integrated Development Environment

To help programmers develop software written in Eiffel, Eiffel Software develops and distributes an integrated development environment (IDE) called *EiffelStudio* which is available on most popular platforms[2]. As other IDEs, EiffelStudio includes a combination of tools integrated under a single user interface, such as

- A compiler
- An interpreter
- A debugger
- An object browser
- A set of metrics tools for code analysis
- A profiler
- A diagram tool

[1]For a detailed treatment of the concepts and theory of the object technology that led to Eiffel's design, see Meyer [3].

[2]EiffelStudio is available on Windows, Linux, FreeBSD, OpenBSD, Solaris and Mac OS

- A set of automated testing tools

The user interface (UI) rests on a number of specific UI paradigms, in particular "Pick-and-Drop"[3] for effective browsing. This is one of the key features that distinguishes EiffelStudio from other similar IDE's, like Microsoft Visual Studio or Eclipse.

The compiler uses a specific trademarked compilation technique known as the *Melting Ice Technology*^{TM} which integrates compilation properly with interpretation of the elements changed since the last compilation. This allows very fast turnaround as recompilation time is proportional to the size of the change, not the size of the overall program. Although such "melted" programs can be delivered as they are (e.g. their *workbench code*), the common practice is to perform a "finalization" step before releasing the software. In short, the finalization is a highly optimized form of compilation which takes longer, but generates optimized executables[4].

The current compiler is able to generate either C or .NET CIL (Common Intermediate Language) code.

1.1.2 Why bother using Eiffel?

Looking at the demands asked of IT-systems being developed today, an increased growth of both size and complexity of such seems to be inevitable. Considering Eiffel, one of the key assets is that it's particularly aimed towards large and complex systems. Eiffel is used as a focal point for software developement within a number of major organizations, i.e. in the financial and security industry, government and national defence, real-time, telecommunication and other industries for mission-critical developments. Thus, wider application of Eiffel can be of global interest.

1.2 The Java Virtual Machine as a target platform

With the argument in place regarding why it is interesting to pay attention to Eiffel, the next question is why should the focus be on targeting the JVM as a platform for Eiffel?

To answer this question, a closer look on the benefits that the JVM can provide is needed and some of these are listed below.

JVM provides:

[3]See `http://www.sonycsl.co.jp/person/rekimoto/pickdrop/`
[4]See `http://en.wikipedia.org/wiki/EiffelStudio`

- A platform which is largely supported by a range of different operating systems
- A large number of libraries available which can be reused for different software projects
- A host to a great diversity of modern languages

In the following, these benefits will be further elaborated.

1.2.1 The JVM as a platform

The two first benefits characterizes the use of JVM as a platform, or more general the *Java platform*[5]. When the Java platform was first introduced by Sun Mircrosystems (now Oracle) back in the start 90's, the main idea sought was to provide a common platform not specific to a certain processor architecture or operating system (OS), that would allow programs written in the Java language to be executed identically. This idea led to what today is known as the heart of the Java platform, the concept of a *virtual machine* that executes JVM bytecode programs, the *JVM*. Along the path of becoming OS independant Sun Microsystems realized that the Java platform could not simply rely on any of the pre-existing OS libraries, as these are radically different from one another. Thus, the Java platform had to provide a comprehensive set of its own standard class libraries containing much of the same reusable functions commonly found in modern OS. Most of these libraries were in fact written in the Java language. Many developers, both inside and outside of Sun Microsystems, have since developed and distributed several thousands reusable libraries for the Java platform, making it one of the largest platforms used in modern software development.

This lead to the realization that for most programming tasks, both trivial and non-trivial, programmers will quickly be able to find a library that can help them achieving the desired result of a feature found in a software product they are working on. They do not have to go about re-inventing the wheel, since chances are that someone have already solved that particular problem and shared the functionality to do so.

1.2.2 The polyglot

All too often the JVM is just associated with the Java language. Indeed, the JVM was originally designed to host and run programs written in the Java language, but the language (Java) and the runtime (JVM) are two seperate units. In fact, any compiler able to translate

[5]Consisting of the language (Java) and the run-time environment (JVM)

and output its source language into JVM bytecode as the target language will be able to be hosted by the JVM.

Today a broad spectrum of languages are supported by the JVM, some of which are existing languages while others are extensions of the Java language. To list a few of the currently supported languages, these include:

- Clojure
- Groovy
- JRuby
- Scala

What is important to note is the fact that the Java platform keeps growing in size. Attracted by a generally available, machine-independent platform, implementors of other languages can turn to the JVM as a delivery vehicle for their languages. To have a greater selection of languages available on the Java platform lead to the attraction of more programmers using different paradigms. For instance, a Java programmer focusing on the object-oriented paradigm may easily work together with a Scala programmer who may be using a functional paradigm. Additionally, as mentioned above, programmers using a language different from Java can benefit from the vast amount of libraries available.

1.3 Related work

Prior to this thesis exists only a few, yet interesting, projects which are directly related to the work behind this report, these are *Eiffel on .NET,* SmartEiffel and *JEiffel*. In the following, the mentioned projects are briefly described.

Eiffel on .NET

The current Eiffel Software compiler is able to generate programs which can run on the .NET platform. In 2002 an article (Simon et al. [4]) was published describing the implementation and integration of the Eiffel language, including DbC, multiple inheritance, genericity, and other advanced facilities, into the Microsoft .NET Framework.

The overall goal for this project was not to just compile Eiffel to .NET CIL, but also to provide a general-purpose framework for multi-language interoperability. The project originates from an earlier experimental project called Eiffel#[6] which featured a partial version of the Eiffel language on the .NET platform. Today the .NET platform is actively supported, maintained and further developed by the Eiffel Software team.

SmartEiffel

Parallel to the Eiffel Software compiler, the SmartEiffel compiler features a free, open-source compiler that translates Eiffel code to either C or JVM bytecode. The SmartEiffel, or SmallEiffel as it originally was named, compiler project was initially started by Domonique Colnet in 1994.

Prior to the translation of Eiffel to JVM bytecode, Domonique Colnet and Olivier Zendra published an article (Colnet and Zendra [5]) describing their strategies for targeting the JVM with genericity, multiple inheiritance, assertions and expanded types. In may 2005, after divergences with the working group for the normalization of the Eiffel language, the SmartEiffel team announced that they would not implement and conform to the Eiffel standard[7]. This decision meant that there was no certainties whether standard Eiffel code would be able to compile on the SmartEiffel compiler henceforth. As of today, the SmartEiffel project is terminated[8].

JEiffel

Benno Baumgartner (ETH, Eidgenössische Technische Hochschule Zürich, Chair of Software Engineering) began the work of a Java back-end extension to the Eiffel Software compiler back in 2005, as described in his thesis (Baumgartner [1]).

For Benno, the goal was not just to compile Eiffel to JVM bytecode and use the JVM as a machine, the goal was also to map Eiffel as closely as possible to Java to allow a tight integration of the two languages (i.e. the same goal as for the .NET back-end). This is different from the approach taken by Colnet and Zendra in the sense that their JVM bytecode generator does not map type relations to Java, but flatten each Eiffel class and generates one Java class out of this.

However, the work was not completed and, as of today, the whereabouts of the source code accompanied the project is unfortunately unknown. But not everything is lost as the report contains the strategies used to map the various Eiffel constructs, including multiple

[6]http://c2.com/cgi/wiki?EiffelSharpLanguage
[7]http://en.wikipedia.org/wiki/SmartEiffel
[8]http://liberty-eiffel.blogspot.dk/2009/09/lets-turn-towards-future.html

inheritance, onto the JVM. Where possible, the strategies found in this report should be used as a foundation for the work in this project.

1.4 Project Goals

The goal of this project is, based on the work of JEiffel and the current .NET CIL back-end, to extend the current Eiffel Software compiler so that it support JVM bytecode as a back-end allowing it to compile programs written in Eiffel to JVM bytecode, with the constraint that the generated code must be able to run on a standard JVM. Furthermore, Java programmers must be able to use these generated classes in their Java programs. Listed below are the prioritized goals for this project:

Primary

1. Extend the Eiffel Software compiler with a JVM bytecode backend, using the work of Benno Baumgartner as a foundation.
2. The JVM bytecode backend must have support for the full Eiffel language.
3. Extend the Eiffel Software compiler front-end to target the JVM, thus allowing users to compile directly to JVM bytecode.
4. Make no decisions to prevent any future multi-language interoperability between the Java and Eiffel programming language.

Secondary (optional)

5. Ensure multi-language interoperability in terms of inheritance.
6. Ensure multi-language interoperability in terms of feature/method calling.
7. Extend the UI in EiffelStudio, such that users can set the JVM as target upon project creation / settings.
8. Extend EiffelStudio for interfacing with the JVM debugger to debug Eiffel applications targeted to JVM.

Chapter 2

Fundamentals

2.1 The Java Virtual Machine (JVM)

A virtual machine (VM) is a software implementation of a machine (i.e. a computer), which executes programs like a real hardware machine. Virtual machines can be separated into two major categories; system VMs and process VMs (sometimes also refered to as an application VM). The Java virtual machine (JVM) falls under the latter of the two categories. A process VM runs as a normal application inside a host OS and supports a single process. The VM is created when a process, or *program*, capable of running inside the VM is started, and destroyed when the process terminates. The purpose of using a process VM is to provide a *platform-independent* environment that abstracts out all details regarding the underlying hardware and/or OS. This means that it is possible to execute a program in the same way on various platforms.

In the following sections the focus will be set on the JVM.

2.1.1 Do you speak the Java language?

Knowing the Java language will aid the understanding of some of the most important concepts in the JVM, as the JVM was originally designed as a platform for running Java programs. A Java program is a collection of class definitions written in the Java language and the Java compiler translates (or *compiles*) the Java program into a format the JVM understands. This compiled form of the Java program is a collection of bytes, represented in a form called the `class` file format[9] which contains *instructions*. When a JVM reads and executes these instructions, the `class` file has the same *semantics* as the original Java program.

Although the JVM was originally aimed at running compiled Java programs, it is theoretically possible to design a translator for any given programming language thus bringing it into the JVM world. As of today a rich set of programming languages is able to run on the JVM.

2.1.2 What is the Java Virtual Machine?

The JVM is an abstract machine and includes a set of standard libraries developed by Sun Microsystems (now Oracle) since 1994 (Lindholm et al. [6]).

[9]The JVM itself knows nothing about the Java language, only of this particular format. This means that any language with functionality that can be expressed in terms of a valid `class` file can be hosted and executed by the JVM.

One goal of the JVM is making the concept of “write once, run anywhere” possible. In order to do so, the Java compiler translates programs written in Java into an intermediate, platform-independent, language called JVM bytecode which is executed by the JVM. This means that as long as a computer has a JVM installed, a program written in JVM bytecode can run on it, independent of the operating system and hardware of the computer. There is also a Just In Time (JIT) compiler within the JVM. The JIT compiler translates the JVM bytecode into native processor instructions at run-time and caches the native code in memory during execution. The use of a JIT compiler means that Java applications, after a short delay during loading and "warm-up", tend to run about as fast as native programs.

The JVM itself it built around a few set of constructs:

- A set of instructions and a definition of the meaning of those instructions. These instructions are called *bytecodes*.
- A binary format called the `class` *file format*, which is used to convey bytecodes and class infrastructure code in a platform-independent manner.
- A *verfication* algorithm used to identify that a program cannot compromise the integrity of the JVM.

Instruction Set

The executable programs running on the JVM are expressed in terms of instructions, called *bytecodes*. The instructions are seen as a set and are designed around a *stack-based* architecture with special object-oriented instructions, which resembles the common sort of instruction sets one would find in most computer architectures.

A JVM bytecode consists of an operation code (opcode[10]), possibly with some arguments following each instruction. Opcodes expect to find the stack in a given state, and transform the stack, so that the arguments are removed and results placed there instead. Each opcode is denoted by a single-byte value, so there are at most 255 possible opcodes (Evans and Verburg [7, chap. 5]). As this list is can be exhaustive to remember, most opcodes fit into one of a number of families as described below:

Load and store opcodes The load and store opcodes is concerned with loading and store values onto the stack.

[10]An opcode is the portion of a language instruction that specifies the operation to be performed.

Arithmetic opcodes The arithmetic opcodes perform some computation on the values on the stack. They take arguments from the top of the stack and perform the required computation, thus returning the result onto the stack.

Execution control opcodes The execution control opcodes are used to perform control-flow, e.g. branching and jumps.

Invocation opcodes The invocation opcodes handles general method calling (including instance and static methods).

Platform operation opcodes The platform operation opcodes includes opcodes for allocating new objects, as well as thread-related opcodes.

The instructions are stored in the `.class` file in a binary format. This makes it quite easy and fast for computers to read, but unfortunatly quite hard for humans to understand. In order to overcome this complication an *assembly language* (a higher level abstraction of the instruction set) for the JVM has been created which can be used to express the content of a `.class` file as human-readable code. An *assembler* is used to translate the assembly code into a binary format.

`.class` file format

The *Java Virtual Machine Specification* (Lindholm et al. [6]) specifies a binary format called the `.class` file, which represents a Java class as a stream of bytes. This means that for each *.java file, a corresponding *.class file must be constructed. The `.class` file contains the compiled JVM bytecode along with a symbol table, as well as other ancillary information.

Verification

In order to ensure that certain parts of the computer is kept safe from tampering, the JVM has a verification algorithm that checks every class and its content. The purpose of this is to ensure that programs follow a set of rules, that are designed to protect the security of the JVM. Looking at a standard C program various exploits can be used to corrupt, or gain access to other areas of the memory[11]. The verification algorithm ensures that this does not happen by tracing through the code to check every object and their usage.

[11]In computer security and programming this technique is best known as *buffer overflowing*.

2.2 Eiffel - An Introduction

The following section presents the Eiffel language shortly[12]. The purpose of this section is to introduce the important concepts of the Eiffel language used throughout this thesis. Familiarity with the concept of object-oriented programming is a necessity.

2.2.1 Getting acquainted with the vocabulary

The first thing to notice is that Eiffel uses a vocabulary that differ some from the one used in other object-oriented languages like Java. This section sets up the vocabulary with references to the terminology used in the Java language (for a quick language terminology comparison, please refer to table A.1 on page 76).

The structure of an Eiffel program

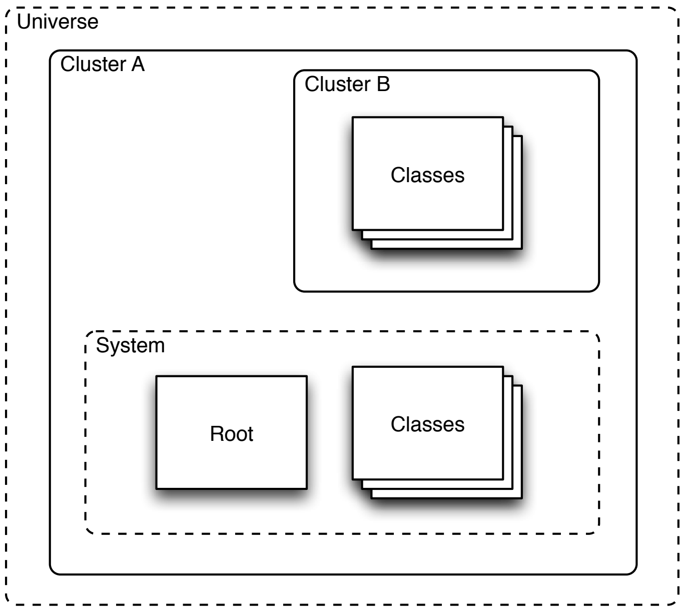

Figure 2.1: The structure of an Eiffel program.

In Eiffel, the basic unit of a program is the *class*. Eiffel does not have a notion of packages like in the Java world. To give an concrete example, suppose one has created a small set of imaging utilities in Java which is to be shared with other people as a common set of tools. Normally one would wrap up the code in a *library* that would result in producing a distributable JAR (*J*ava *Ar*chive) file other developers could *include* and *import* in their own Java projects. In Eiffel, one don't import the libraries included, one simply use them.

[12]For a more detailed presentation, please refer to Meyer [8]

An Eiffel *system* is a set of classes (typically a set of *clusters* containing classes) that can be used to assemble and produce an executable. The system only includes the classes which are needed for the program execution. Clusters are not a syntactic language construct, but rather a standard organizational convention. An Eiffel program is organized with each class in a separate file and each cluster is a directory containing these class files. In this organization, subclusters corresponds to subdirectories.

In order to represent all classes of a system, needed during program execution or not, Eiffel uses the notion of an *universe*. A universe is a superset of the system, and corresponds to all the classes present in the clusters defined in the Eiffel system, even if they are not needed for the program execution. For a graphical overview, please refer to figure 2.1 on page 10.

To define the entry-point of the program Eiffel uses the notion of a *root class*. This class is the first class that is instantiated during program execution using its creation procedure known as the *root creation procedure*. An Eiffel system corresponds to all the classes that are needed either directly or indirectly by the root class (i.e. the classes that are reachable from the root creation procedure).

The definition of what a Eiffel system contains is described in an *Ace file*, which is an configuration file written in a language called *Language for Assembly Classes in Eiffel* (LACE). An Ace file (or section) specifies:

- The root class and the root procedure (only for programs).
- The options used for compilation of the system.
- The used clusters and libraries along with their options.

Classes and Types

Like in Java, Eiffel denotes a class to a representation of an Abstract Data Type (or ADT for short). In Eiffel every *object* is an instance of a certain class. The creation, or instantiation, of an object uses a so-called *creation procedure*, which is similar to the notion of a "constructor" found in Java.

A class is characterized by a set of *features*, which falls under the categories of being either an *attribute* or a *routine*. This is similar to what is found in Java, having fields and methods. Yet, Eiffel further distinguishes between routines that returns a result (*functions*) and routines that does not return a result (*procedures*). This classification leads to an important aspect of the Eiffel language, i.e. the *Command/Query Seperation principle* where features can either be *commands* (if they *do not return* a result) or *queries* (if they *do return* a result) - a feature should not both change the object's state and return a result about this object.

As mentioned, in Eiffel, every object is an instance of a class. Eiffel is a strongly typed language. As in Java, even basic types like `INTEGER`'s or `BOOLEAN`'s are represented as a class. The majority of the types found in Eiffel are *reference types*, which means that the value of a certain type is a reference to an object, not the object itself.

However, there is also a second category of types, called *expanded types*, where the value is the actual object. In Eiffel basic types are an example of the latter type. This means that a value 42 of type `INTEGER` is indeed an object of type `INTEGER` with value 42, and not a reference to an object of type `INTEGER` with a field containing the value 42.

Design By Contract™

The concept of DbC is the center piece of development in Eiffel. The contracts assert what must be true when a routine *is executed* (a pre-condition) and what must hold to be true after the routine *has been executed* (post-condition) using boolean expressions. In addition, class invariant is used to define assertions which must hold true before and after any feature is called (both attributes and routines). Furthermore, the language supports a *check instruction* (a kind of "assert") and loop invariants.

If presented in the right form, the concept behind DbC is not something that should be unfamiliar. As said in Meyer [9]:

> “ *This idea of contract defined by some obligations and benefits is an analogy with the notion of contract in business: the supplier has some obligations to his clients and the clients also have some obligations to their supplier. What is an obligation for the supplier is a benefit for the client, and conversely..* ”

2.3 Revisiting Compiler Concepts

A compiler is a program that takes as input a program written in one language (the *source* langauge), and translate it into a program in another language (the *target* language). The source language is often a high-level language like Eiffel or Java, and the target is usually a low-level language like assembly or machine code. The most common reason for wanting to transform source code is to create an *executable* program.

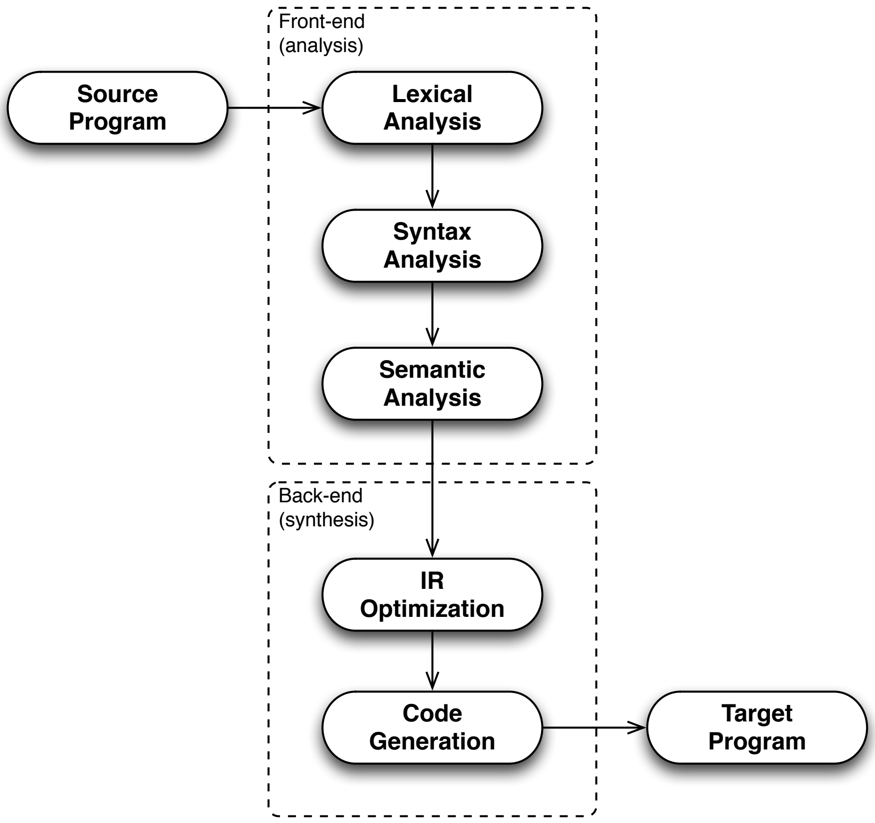

Figure 2.2: Diagram of a common compiler structure.

How does a compiler work?

Looking at figure 2.2 it is seen that there are two main stages in the compilation process: *analysis* (front-end) and *synthesis* (back-end). The analysis stage breaks up the source program into pieces (*lexical analysis*) and creates a generic intermediate representation (or IR for short) of the program (*syntax analysis*). An example of this IR could be something like an Abstract Syntax Tree (AST). Once the IR has been build, a *semantic analysis* takes the representation made from the syntax analysis and applies semantic rules to the representation to make sure that the program meets the semantic rules requirements of the language (*type checking*). The final stage converts the IR of the program into an executable set of instructions (often assembly) (Aho et al. [10]).

Code Generation

The final stage of the compilation consist of converting the IR into a set of instructions, thus allowing the program to be executed by a machine. This process is also known as *code generation*. The input to the code generator is the IR of the source program produced by the front-end of the compiler.

Once the AST has been passed as input to the back-end it typically undergoes several changes where the IR is optimized and more data is annotated to it (like debug information) before it is passed further along to the actual code generator. From this new highly optimized IR, the actual generation of the new target language takes place.

As focus will be on the code generation when implementing the JVM back-end, it is henceforth assumed that the front-end has analysed and translated the source program into an IR. It is also assumed that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type- and conversion operators have been inserted wherever necessary. The code generator can therefore safely proceed on the assumption that the input given to it is free of any kinds of these errors (Aho et al. [10]).

Chapter 3

Analysis

3.1 From Eiffel to JVM Bytecode

The code generation phase centers around the optimized IR. When it comes to the actual translation of the IR to target language, in this case JVM bytecode, one may refer to the translation pattern known as *model-driven translation*, see figure 3.1 (Parr [11, chap. 11]).

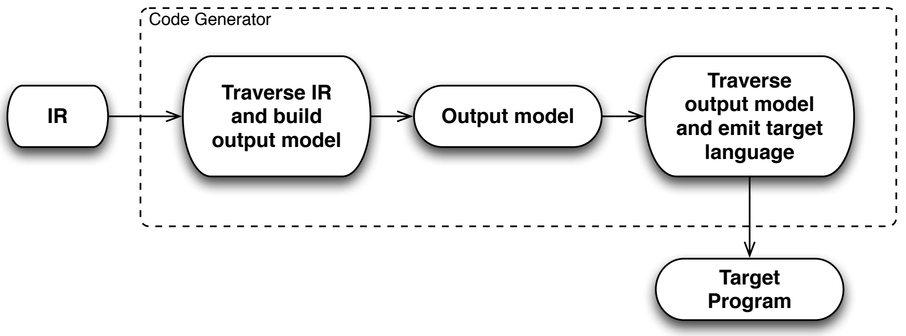
Code Generator
IR
Traverse IR and build output model
Output model
Traverse output model and emit target language
Target Program

Figure 3.1: Generator for model-driven translation.

In its simplest form, a model-driven translation takes an IR as its input and then traverses it to generate an *output model* which represents the desired target language (Parr [11]). The structure of the output model is always a nested arrangement of output objects (Parr [11]). As the input model is being traversed, objects representing each phrase being translated must be created, thus mapping the input model to the output model. When the output model has been constructed it can be used to generate the executable target program.

In the following sections, focus will rest on how to derive an appropriate output model corresponding to the JVM `.class` file format. In order to do so it is first needed to have a basic understanding on the internal structure of a `.class` file. From here focus will shift towards the mechanics behind constructing the output model.

3.1.1 Anatomy of the `.class` file format

The overall structure of a `.class` file contains most of the structural information similar to what would be found in Java source code (Lindholm et al. [6], Bruneton [12]), as seen on figure 3.2. The `.class` file contains:

- A section describing the access modifers[13] (such as `public` or `private`), the class name, the super class along with the interfaces the class is implementing.

[13]Access modifiers (flags) are found on classes, fields and methods. For more details on the specific access modifiers supported on each type, please refer to Lindholm et al. [6, chap. 4].

- A section per field declared in the class. Each section describes the modifers, the field name and type of the field.
- A section per method and constructor declared in the class. Each section describes the modifers, the method name and the return- and parameters types. It also contains the body of the method, in the form of a sequence of JVM bytecode instructions.

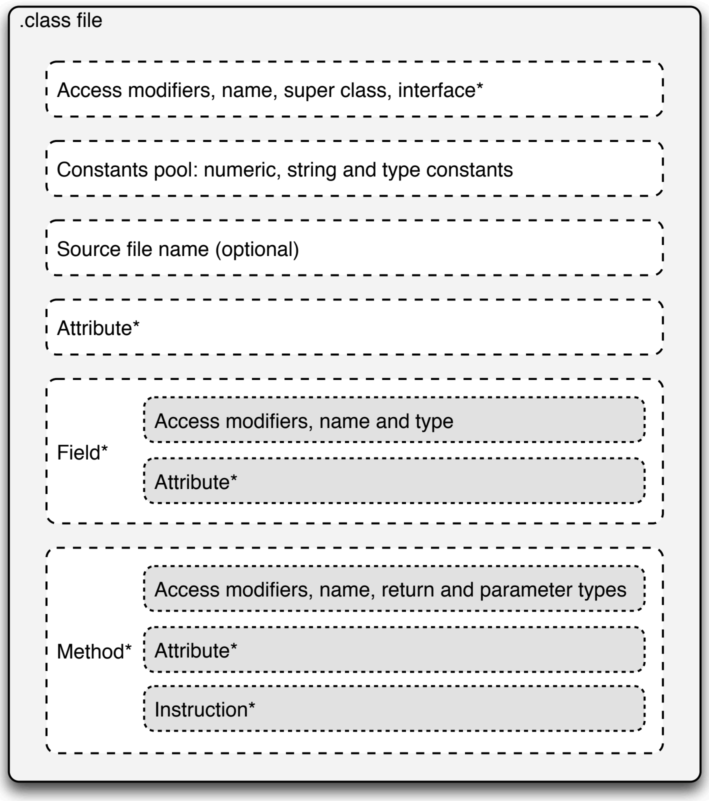
.class file
Access modifiers, name, super class, interface*
Constants pool: numeric, string and type constants
Source file name (optional)
Attribute*
Field*
Access modifiers, name and type
Attribute*
Method*
Access modifiers, name, return and parameter types
Attribute*
Instruction*

Figure 3.2: An overview of the `.class` file (* means zero or more).

Comparing a `.class` file to a Java source code file, there are some important difference to be aware of:

- A `.class` file contains only one class, while a Java source code file can contain several classes (e.g. inner classes).
- A `.class` file does not contain comments, but can contain class, field, method and code `attributes` used to associate additional information to these elements.
- A `.class` file does not contain a `package` or `import` section, which means that all type names must be *fully qualified*.

Another important structural difference is that a `.class` file contains a *constant pool* section, which contains field descriptions and method descriptions, string constants, large integer constants, and so on. These constants are only defined once in the constant pool, thus referenced by their index when needed (Lindholm et al. [6], Bruneton [12]).

The last important difference between a `.class` file and a Java source code file is the way Java types are respresented. In the next few sections type representations in `.class` files will be explained.

Internal naming

In `.class` files types are represented with an *internal name*. The internal name of a class is simply the fully qualified name of the class, where the dots have been replaced by slashes (Lindholm et al. [6], Bruneton [12]). To give an example of an internal name, the qualified name of the `String` class found in Java would be translated as follows:

$$java.lang.String \rightsquigarrow java/lang/String$$

Type descriptors

Internal names are used only for classes and interfaces, in all other situations, such as field types and Java types, these are represented in the `.class` file using *type descriptors* (see table 3.1).

Java type	Type descriptor
`boolean`	`Z`
`char`	`C`
`byte`	`B`
`short`	`S`
`int`	`I`
`float`	`F`
`long`	`J`
`double`	`D`
`Object`	`Ljava/lang/Object;`
`int[]`	`[I`
`Object[][]`	`[[Ljava/lang/Object;`

Table 3.1: Type descriptors of a subset of the Java types.

The type descriptors of primative types are denoted single characters, where the descriptor of a class type is the internal name of this class, preceded by L and followed by a semicolon. Finally the descriptor of an array type is a square bracket followed by the descriptor of the array element type (Lindholm et al. [6], Bruneton [12]).

Method descriptors

A *method descriptor* is a list of type descriptors that describe the parameter types and the return type of a method, in a single string (Bruneton [12, p. 12]). A method descriptor starts with a left parenthesis, followed by the type descriptor for each formal parameter, followed by a right parenthesis, followed by the type descriptor of the return type, or `V` if the method returns `void` (Bruneton [12, p. 12]).

Method declaration	Method descriptor
`void m(float f, int i)`	`(FI)V`
`int m(Object o)`	`(Ljava/lang/Object;)I`
`int[] m(int i, String s)`	`(ILjava/lang/String;)[I`
`Object m(int[] i)`	`([I)Ljava/lang/Object;`

Table 3.2: Method descriptors for some Java method declarations.

It should be noted that the method descriptor *does not* contain any information about the method's name, nor the names of the formal parameters in its declaration, as this information is not needed by the JVM. Table 3.2 gives a few examples of some method descriptors.

3.1.2 Towards defining the Target Output Model

With a basic understanding on the internal structure of the `.class` file format, the next concern is to define the *target output model*. In the world of object-oriented programming, it is usually customary to write classes to solve a perticular set of problems, in this case deriving a proper output model for the `.class` file format.

Following an object-oriented approach, assume that one wants to generate JVM bytecode for a standard "Hello, world!" program written in Java. In order to do so varies tools for generating JVM bytecode can be found in the Java world, such as the Byte Code Engineering Library (BCEL) or ASM (A Java bytecode engineering library). For example, listings 3.1 shows how to generate the program using the ASM[14] library.

Listing 3.1: A HelloWorld program written using ASM.

```java
// Creates a ClassWriter for the HelloWorld public class,
// which inherits from Object
ClassWriter cw = new ClassWriter(0);
cw.visit(V1_1, ACC_PUBLIC, "HelloWorld", null, "java/lang/Object", null);

// Creates a MethodWriter for the 'main' method
mw = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
          "([Ljava/lang/String;)V", null, null);
// Pushes the 'out' field (of type PrintStream) of the System class
mw.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
          "Ljava/io/PrintStream;");
// Pushes the "Hello, world!" String constant
mw.visitLdcInsn("Hello, world!");
// Invokes the 'println' method (defined in the PrintStream class)
mw.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println",
           "(Ljava/lang/String;)V");
mw.visitInsn(RETURN);
// This code uses a maximum of two stack elements and two local variables
mw.visitMaxs(2, 2);
mw.visitEnd();

// Gets the bytecode of the Example class, and stores it into the
// HelloWorld.class file
byte[] code = cw.toByteArray();
FileOutputStream fos = new FileOutputStream("HelloWorld.class");
fos.write(code);
fos.close();
```

Without going into the details of the code listed in listings 3.1, the overall steps in generating the `HelloWorld.class` file can be summarized as follows:

1. Creating the class object, which represents the `.class` file to generate.
2. Creating each method to be put into the `.class` and append it to the class object.
3. Emit the bytecode of the class object into a physical `.class` file using a file-stream.

The approach of using a library for generating Java bytecode such as ASM, these so-called *Target-Specific Generator Classes* (Parr [11, p. 308]), works well and feels comfortable.

[14]See `http://asm.ow2.org/`

However, there are two problems related to the use of such a library. The use of a library, such as the ASM, comes at a cost as it is often quite cumbersome to create such large output structures manually.

Looking at listings 3.1 it seems to require a great amount of work to create a very simple program, so often it may be easier to just emit output directly while traversing the input model (Parr [11, p. 287-289]). Secondly, no such library exists in Eiffel. Refering to the work on JEiffel (Baumgartner [1]), JVM bytecode was generated using a Eiffel library to generate JVM bytecode (Gisel [13]). As with the source code of the work on JEiffel, neither does the source code of this JVM bytecode library exist[15].

The problem of creating a large output structure can be perceived differently. If the input model *and* output model shares a substantial amount of similarities, and if the target language has a well-defined assembly language, then one could emit the input model directly to the assembly language. It can be more effective to emit the output without the hassle of first building up the output structures, but in the case where the input model can not be directly translated to its output model, this approach falls short.

In the case of translating Eiffel to Java (or JVM bytecode), the problem is that Eiffel has a richer object model then the one JVM exposes, which involves multiple-inheiritance, agents (closures) and rescue clauses[16].

Additionally, since Oracle have not defined an assembly language for the JVM[17] one could end up emitting JVM bytecode that would either render obsolete or, in worst case, change too much in structure to be reused when, and if, Oracle at some point in time does indeed define an assembly language.

As for the second problem, this leads to either of the following:

A Review whether is is possible to use one of the JVM bytecode libraries from within Eiffel.

B Development of a new JVM bytecode library in Eiffel.

Considering option (A) there are two possible ways to achieve this, either Eiffel must be able to call Java code externally (as with calling C code externally from Eiffel), or one of the existing bytecode libraries must be translated from Java source code to Eiffel.

It is indeed possible to call Java methods and fields from Eiffel code using the *Eiffel2Java*[18]

[15] An extensive search for this library was conducted early on in the project, but without luck.

[16] Which is indeed more complex then a simple `try-catch` block found in Java.

[17] In contrast to Microsoft which have a well-defined assembly language for the .NET platform, i.e. CIL.

[18] See `http://docs.eiffel.com/book/solutions/eiffel2java`

interface which uses the Java Native Interface (JNI) provided by the Java Development Kit (JDK). Following the path of using Eiffel2Java would involve creating a *wrapper* around either ASM or BCEL in Eiffel, thus hiding and abstracting out the unnecessary details of calling the JVM bytecode libary through the Eiffel2Java interface thereby giving the clients of the Java bytecode library an "easy-to-use" interface to work against when generating JVM bytecode. The benefit of using this approach is that it is possible to reuse an already well-defined, tested set of widely used library functions able to generate `.class` files, hence JVM bytecode. The drawback of using Eiffel2Java is that is introduces the complexity of having to deal with allocating and managing the resources of an external library. Furthermore, in its current state of the documentation for this library leads the client to believe that this is still a research project which is by to end completed. In addition, translating one of the JVM bytecode libraries source code to Eiffel is also a possibility using the utility described in Trudel et al. [14]. However, there might be some restrictions to wheter this is possible due to licences of the ASM and BCEL libraries. Moreover, due to the limitations mentioned in Trudel et al. [14], the translation is not fully implemented, thus the translated library could end up not turning out as aspected.

As for option (B) a subset of the JVM bytecode libraries found in Java, able to generate just the necessary JVM bytecode required in this project, could be developed in Eiffel. Writing a new library will eliminate the problems introduced by option (A), as well as contribute to the Eiffel Software community with a reusable library which can be further developed. However, the problem related to implementing such a library is that it requires having to deal with the details of the `.class` binary file format to a large extend. This can be rather cumbersome since a great amount of knowledge on the philosophy of the JVM is needed before one might be able to write something at the JVM level and generate a `.class` file. Fortunately, an assembly language does exists for the JVM, named *Jasmin*[19], which may be a simpler approach than writing a Java `.class` file generator, as this leaves out getting into the details of constant pool indices, attribute tables, and so on.

The Jasmin Assembly Language

Jasmin is a Java Assembler Interface (JAI) that takes an ASCII description of a Java class, written in an assembler-like syntax (e.g. the Jasmin Assembly Language), and translates it into `.class` files suitable for loading into the JVM. Listings 3.2 on page 22 gives an idea on how a HelloWorld program looks like written in Jasmin assembly code. The structure[20]

[19]SourceForge Open Source project found at `http://jasmin.sourceforge.net/`

[20]For a detailed overview of the Jasmin assembly file format structure, please refer to Appendix B.

of a Jasmin assembly code file, based on the Jasmin manual[21], consists of a sequence of newline-seperated statements.

There are three kinds of statements in Jasmin:

Directives: Directive statements are used to give Jasmin meta-level information to the assembler. A directive statement consists of a directive name followed by zero or more parameters separated by spaces, thus a newline.

Instructions: An instruction statement represents a standard mnemonic JVM opcodes, which consists of an instruction name, zero or more parameters separated by spaces, and a newline.

Labels: The label statements can be used within method definitions upon *branching*. A Jasmin label statement consists of a name followed by a ':', and a newline.

Listing 3.2: The canonical HelloWorld example in Jasmin.

```
.class public HelloWorld
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
  .limit stack 2
  .limit locals 2

  getstatic      java/lang/System/out Ljava/io/PrintStream;
  ldc            "Hello World."
  invokevirtual  java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```

As seen on listings 3.2, a Jasmin assembly code file starts by providing two kinds of directives describing the class, such as the name of the class and the name of the superclass:

- `.class <access-spec> <class-name>`
- `.super <class-name>`

This is also called called the *header area*. The `<access-spec>` specifies the access modifiers, and is a list of zero or more keywords that specifies the *access levels* of the class and other attributes for the class. Additional directives found in the header area includes

[21]See `http://jasmin.sourceforge.net/guide.html`

`.source`, `.interface` and `.implements`, whereas `.source` defines the name of the source file that the class originated from, `.interface` declares a Java interface rather then a Java class and `.implements` specifies a list of the interfaces that are implemented by the class being defined.

After the header information follows a list of of field definitions, also called the *field area*. A field is defined using the `.field` directive:

```
.field <access-spec> <field-name> <descriptor> [ = <value> ]
```

where, `<access-spec>` specifies the access modifiers, the `<field-name>` denotes the name of the field, and `<descriptor>` is the type descriptor of the field. The last part of the directive `[ = <value> ]` is optional, but allows initialization of an integer, a quoted string or a decimal number rendering the field `final`.

At the end follows a list of method definitions, also called the *method area*. A method is defined using the basic form:

```
.method <access-spec> <method-spec> <statements> .end method
```

Here the `<access-spec>` specifies the access modifiers, the `<method-spec>` is the type descriptor of the method, and `<statements>` is the code defining the body of the method. The body of the method consists of JVM instructions[22], which can take zero or more parameters, depending on the type of instruction used[23].

Generating JVM bytecode

Based on the foregoing analysis, an output model should be defined which closely maps to the `.class` file structure described. Upon traversing the IR, i.e. the internal AST of the Eiffel classes in the system being compiled, the output model is to be generated. Once the IR has been traversed, the generated output model is traversed and translated into the Jasmin assembly language.

The final step is to assemble the Jasmin assembly code files into `.class` files using the Jasmin assembler. Once the Jasmin assembly code files has been translated, the `.class` files are ready to be used in the JVM.

[22]Please refer to Lindholm et al. [6, chap. 6]

[23]For a full list of supported instructions available in Jasmin, please see `http://jasmin.sourceforge.net/instructions.html`

3.1.3 Alternative ways of generating JVM bytecode from Eiffel

As Eiffel is compiled down to C/CIL, one could do a direct translation of the binary output to JVM, i.e. a machine translation, going from binary to binary. An example of such can be found in the C# to Android project, where the development team of Mono wanted to be able to run code written in C# on the Android platform (Ward [15]). Another attempt on translating .NET to JVM using a binary translation can be found in Lee and Na [16].

Doing a direct binary to binary translation does come with its own set of drawbacks. For instance, doing a binary to binary translation of C to JVM bytecode may result with having a generated `.class` file that would be near impossible to use for language interoperability between Eiffel and other JVM languages, e.g. Scala. Looking at translating .NET to JVM bytecode may also present some difficulties, even though .NET and the JVM platform share a substantial amount of the same ideas. The first problem is that the .NET platform presents a richer bytecode model, for instance it has generics embedded in its bytecode language, the JVM does not. Secondly, the current .NET back-end of the Eiffel compiler would not in its existing state be able to run on UNIX, as it is bound to Win32 APIs. So even if one did do translation from .NET to Java, this would only work under the circumstances that the programmer was running on a Windows platform.

3.2 Mapping Eiffel onto the JVM Object Model

The following sections presents a translation model for Eiffel to Java. The general schema is that an Eiffel construct is introduced by an example, where necessary, thus the example in Eiffel is translated to its conterpart in Java. The assumption is, that if a Eiffel construct can be translated to Java, the it can be translated into JVM bytecode. As for the latter, implementors of a JVM back-end using the EJBCL API may compile the examples shown in the following sections into `.class` files and then decompile the generated `.class` file(s) using *D-Java*[24] with "`-o jasmin`" flag to get the equivalent Jasmin assembly code file(s).

3.2.1 Type Mapping

As described in Colnet and Zendra [5] and Baumgartner [1], the basic Eiffel types should be mapped directly to the corresponding JVM bytecode, according to the following correspondence table:

Eiffel type	JVM bytecode type
`INTEGER`	`I`
`REAL`	`F`
`DOUBLE`	`D`
`INTEGER_8`	`B`
`INTEGER_16`	`S`
`INTEGER_64`	`J`
`CHARACTER`	`B`
`BOOLEAN`	`Z`
`POINTER`	`Ljava/lang/Object;`

Table 3.3: Type mapping between Eiffel and JVM bytecode.

Moreover, the normal Eiffel types should be mapped to reference types in Java. This means that each Eiffel reference type should be mapped to one specific Java `.class` file. A class named `HELLO_WORLD` is thus coded in an `hello_world.class` file. The same translation scheme is used for Eiffel library types, which are reference types, like `STRING`.

[24]See `https://www.vmth.ucdavis.edu/incoming/D-Java/djava.html`

3.2.2 Multiple-Inheritance

Java is a single-inheritance language meaning that each class is based on exactly one other class: the derived class *extends* the it is based on, also called its base class or *super class*. An instance of a derived class may be used any place where an instance of a base class for this type is called. In addition to its own, the derived class has all the behaviors of it's base class. Being single-inheritance language also means that these is no direct support for applying multiple-inheritance to classes.

Although the JVM does not support multiple-inheritance in the case of concreate classes, it does support multiple-inheritance in terms of *interfaces*. Since interfaces does not contain any implementations, they are not subject to the problems of multiple-inheritance, thus interfaces can be used to help modelling multiple-inheritance (Engel [17]).

Using Interfaces to model Multiple-Inheritance

Consider a hypothetical example consisting of three classes, A, B and C. Class C inherits from class A and B, whereas the A and B both define a method called `f`, which is unrelated to one another:

Listing 3.3: Conceptual multiple-inheritance in Java.

```
class A {
  String f() { /* Implementation of f() of A */ }
}
class B {
  String f() { /* Implementation of f() of B */ }
}
class C extends A, B { /* A f() from both A and B */ }
```

For each class, an interface is declared, thus the definition of C inherits multiple interfaces:

Listing 3.4: Modelling multiple-inheritance in Java using interfaces.

```
interface A {
  String f();
}
interface B {
  String f();
}
interface C extends A, B {
}
```

In order to use these interfaces, classes that implement them must be provided.

Implementing the Interfaces

Each implementing class will follow the notion of `<interface-name>$class`. Each class implements the corresponding interface and provides implementions for the method bodies:

Listing 3.5: Implementing the interfaces.

```
class A$class implements A {
  String f() { return "f from A"; }
}
class B$class implements B {
  String f() { return "f from B"; }
}
class C$class implements C {
  /*
   * Problem: What should be done here,
   * as f is coming from both parents?
   */
}
```

The class `A$class` and `B$class` involves no inheritance, so their methods are implemented according to their interface. The problem lies in implementing `C$class`, as it defines no methods of its own, but needs to have all the methods of both `A$class` and `B$class`. Unfortunately, `C$class` can not inherit from both `A$class` and `B$class` due to the restrictions of the JVM.

Fortunately, as discussed in Engel [17, chap. 11, p. 300-302], it is possible to implement `C$class` using neither `A$class` or `B$class` as base classes. As described, `C$class` must instead hold private instances of both `A$class` and `B$class`, called `A$delegate` and `B$delegate`. When a method is called, `C$class` defers the implementation of that method by calling to the equivalent method on either the delegate object for A or delegate object for B. This is also the same translation model used in Baumgartner [1]. The problem related is how the class `C$class` should resolve the conflicting methods with the same name.

Resolving Naming Ambiguities

In Eiffel, class `C$class` would be illegal because of the ambiguity. To resolve this conflict, Eiffel makes use if its *renaming* facility:

Listing 3.6: Resolving naming conflicts through the use of renaming.

```
class A
feature
  f: STRING
    do
      --| Implementation of routine f in class A...
    end
end

class B
feature
  f: STRING
    do
      --| Implementation of routine f in class B...
    end
end

class C
inherit
  A rename f as f_a end
  B
end
```

As shown in listings 3.6, the naming conflict is resolved by renaming the `f` coming from `A` as `f_a`. For the interface of class C, this gets translated into:

Listing 3.7: Implementation of class C in Java using renaming.

```
interface C extends A, B {
  String f_a();
  /* And the additional method f() from B, which was not renamed. */
}
class C$class implements C {
  private A$class A$delegate = new A$class();
  private A$class B$delegate = new B$class();

  public String f_a() { return A$delegate.f(); }
  public String f() { return B$delegate.f(); }
}
```

Where usage of class `C`'s implementation `C$class` yields:

Listing 3.8: Calling the renamed features.

```
C c = new C$class();
c.f();    // yields f of B
c.f_a();  // yields f of A
```

However, as described in Baumgartner [1, chap. 2, p. 10-11], there some subtleties bound to this approach in terms of the binding algorithm used when doing polymorphic dispatching in Eiffel contra the one used in the JVM, which are not trivial. To illuminate this, an example is shown in listings 3.9:

Listing 3.9: Binding in Eiffel when doing polymorphic dispatching.

```
local
  c: C
  l_as: ARRAY [A]
do
  create c
  c.f    --| yields f of B

  create l_as.make (0, 0)
  l_as.put (c, 0)
  l_as.item (0).f    --| yields f of A
end
```

This is rather interesting, because although the two calls to `f` is made on an instance of the exact same object, the executed code is not the same. This is due to the fact that in Eiffel, not only is the type of an object at run-time being taken into consideration when binding to an implementation, but also the static type of the expression which returns the reference to that object. In the example, the type of `c` is `C`, but the type `as.item (0)` is `A`, and therefore the `f` from `A` is called instead of `f` of `B`.

Trying to do the same in Java yields:

Listing 3.10: Binding in Eiffel when doing polymorphic dispatching.

```
C c = new C();
c.f();    // yields f of B

A[] as = new A[1];
as[0] = c;
as[0].f();  // yields f of B
```

Here the second call to `f` does not result in the same behavior as the result shown in listings 3.9. This is not unexpected, since all methods in Java are virtual, and therefore the JVM will always bind to the most recent implementation of the method being called (Lindholm et al. [6]). From the perspective of the JVM, the problem is that it does not know that `f` from `A` is unrelated, thus different from the `f` of `B`. Therefore it is not possible, in the case of renaming, to use the standard JVM dispatch mechanism provided by an underlaying `invokevirtual` bytecode instruction, which is based on fixed routine names. Using `invokevirtual` to do the dispatch, results in a solution with mangled names. The problem with such a solution is the problem related to how one effectively hides the name mangling from the client, which makes the translated model much more complex, as described in Baumgartner [1] and Colnet and Zendra [5].

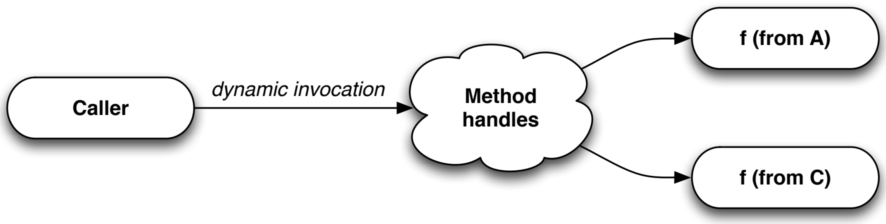

Figure 3.3: Mimicking the the polymorphic feature dispatching found in Eiffel on the JVM, using dynamic invocation and method handles

To overcome the complications described, a possible solution may be to make use of the new bytecode instruction introduced in Java 7, which appears under the name `invoke-dynamic`. This new bytecode instruction enables implementers of a dynamic language to translate a method invocation into bytecode, without having to specify a target type that contains the method. When the JVM sees an `invokedynamic` bytecode instruction, it uses a new *linking mechanism* to get to the method it needs. The new linkage mechanism for dynamically typed languages involves a new structure called *method handles*, that enables the JVM to invoke the correct method in response to an `invokedynamic` bytecode instruction. A method handle is a simple object type that contains an anonymous reference to a JVM method, which is accessed through a *pointer structure*. Using the new linkage mechanism means that the first time an `invokedynamic` bytecode instruction is executed by a specific caller, it is linked. When the linking occurs, a method handle is assigned to the individual `invokedynamic` bytecode instruction as its target. The next time a call is made to the method previously linked by the same caller, the method handle assigned will be used by the JVM, thus redirecting the call to the correct method. The linking between a caller and a method handle happens through a *bootstrap* method which is defined in the class whom may receive a request from a dynamic invocation.

To mimic the the polymorphic feature dispatching found in Eiffel on the JVM, a method handle containing information on the calling type along with the parameters for `f` should be implemented in class `C`, along with a bootstrapper for linking `f` to either the `f` of class `C` or class `A` (see figure 3.3 on page 30). When generating the code for such calls, it should be ensured that one does not call `f` directly on class `C`, but rather a dynamic invocation using the method handle for `f` passing to it the static call type, such as `A` in the example shown in listings 3.9. On calling the method handle for each calling type, either class `C` or class `A`, the bootstrap method will then link the caller to either `f` from `C` or `f` from `A`. Hence, for calling `f` on an item retrieved from an array of type `A`, it should be possible to link the appropriate version of `f` to the call, thus bypassing the standard dynamic dispatching in the JVM.

3.2.3 Constants and Attributes

As interfaces supports having constants defined in the form `final <type> CONST_NAME = <value>`, any constant defined in an Eiffel class can be added directly to the interface. To define attributes of an Eiffel class on an interface, a possible solution is to specify *accessor* methods for each attribute (Engel [17, chap. 11, p. 302-303]). Listings 3.11 shows both how a constant value and an attribute of class A (named attr1) can be implemented:

Listing 3.11: Implementing the constants and attributes using interfaces.

```
interface A {
  final int INTEGER_CONST = 42;
  int getAttr1();
}
class A$class implements A {
  private int attr1;
  public int getAttr1() { return this.attr1; }
}
```

3.2.4 Creating Objects

When creating objects in Eiffel, clients are not allowed to call the creation routines directly without using the `create` keyword, as seen in listings 3.12.The problem with the way Eiffel defines how objects are created is that Java does not allow having named constructors. Constructors must always have the name of the class, where the only way to have multiple constructors is through overloading.

Listing 3.12: Creating an instance of class A in Eiffel.

```
class A
create
  make
feature {NONE}
  make
    do
      --| Initialize...
    end
end

class B
feature
  instantiate_A
    local
      l_a: A
    do
      l_a.make --| Fails, wrong syntax.
      create l_a.make --| Works, correct syntax.
    end
end
```

Moreover, it is arguable whether these creation features should be included in the interfaces, as clients of the interfaces should not, according to the syntax in Eiffel, have direct access to these without first introducing the `create` keyword. In order to deal with both problems, a possible solution is to hide any Java constructors exposed to clients trying to instantiate objects of class A, thus introduce *factory methods* (Gamma et al. [18, p. 107-116]) in each implementation classes that corresponds to the creation features found in the Eiffel classes:

Listing 3.13: Creating an instance of class A in Java.

```
class A$class implements A {
  protected A$class() { }
  public static A make()  { return new A$class(); }
}
class B$class implements B {
  public void instantiate_A() { A l_a = A$class.make(); }
}
```

3.2.5 Deferred classes and features

Listing 3.14: A deferred class with a deferred feature.

```
deferred class A
feature
  u: BOOLEAN
    do
      --| Code of feature u.
    end

  v: INTEGER
    do
      --| Code of feature v.
    end

  w: INTEGER
    deferred
    end
```

The `deferred` keyword has a direct conceptual equivalent in Java, called `abstract` which may contain a partially implementated class. If a class in Eiffel containts one or more features which are deferred, the class must also be deferred, which is similar to the notion of abstract in Java. Furthermore, a deferred class can not be instantiated.

If a class is, or contains a feature which is deferred, a translation to Java should explicitly set the class to be `abstract`. Deferred features should not by defined in the class, as these will appear to be abstract in the interface implemented by the class:

Listing 3.15: The equivalent abstract class and method in Java.

```
abstract class A$class implements A {
  public boolean u() { /* Code of u. */ }
  public int v() { /* Code of v. */ }
}
```

3.2.6 Redefining and Undefining

Redefining a feature in Eiffel means to redefine the content of a feature, while keeping the same signature. In Java, this is known as *overriding* of a method and is therefore built into the Java language.

Upon translating a redefined feature of a class, a new feature should be created in the corresponding Java class which contains the new content of the feature's body.

Listing 3.16: Redefining and undefining features.

```
deferred class B
inherit
  A
    undefine v end
    redefine u end
feature
  u: BOOLEAN
    do
      --| Some code which is different from A...
    end
end
```

Undefining a feature means to take an effective feature of a parent, thus turning it into a deferred feature. The translation of undefined features should yield the same translation scheme as the one defined for deferred features.

3.2.7 Feature Bodies

Translating the code forming the body of the feature in Eiffel to code in Java, follows a standard set of well-known templates which is described in Aho et al. [10]. For example, translating an `if E then S1 else S2` construct to JVM bytecode yields:

Listing 3.17: An example of how to translate an `if-then-else` construct to JVM bytecode.

```
E
ifeq false
S1
goto endif
false:
S2
endif:
nop
```

Translating constructs like expressions, assignments, loops, conditions is not discussed further in this report.

3.2.8 Once routines

In Eiffel it is possible to *lazy-load* both routines and procedures, using the `once` keyword as shown in listings 3.18 on page 35.

Listing 3.18: An example of using once.

```
class SINGLETON
feature
  instance: SINGLETON
    once
      create Result
    end
end
```

As described in Baumgartner [1, chap. 2, p. 36-37], a *singleton* (Gamma et al. [18, p. 107]) can be used to model this behavior. The solution presented in Baumgartner [1], will however meet some resistance if it was to be used in practice. To justify this claim, the singleton implementations used in the solution uses a lazy initialization of the one instance. This means that the instance is not created when the class loads, but rather when it is first used. The solution presented in Baumgartner [1] has the flaw that it neglects the use of synchronization, which can lead to multiple instances of the singleton class (Fox [19]). Introducing syncronization will allow multiple threads to safely execute concurrently on all invocations except the first. However, because the method is synchronized, one pays the cost of synchronization for every invocation of the method, even though it is only required on the first invocation. As more advanced JVMs have emerged, the cost of synchronization has decreased, but there is still a performance penalty for entering and leaving a synchronized method or block (Haggar [20]).

A solution to these problems leads to the *Initialization on Demand Holder (IODH) idiom*[25], the only difference being that `once` routines can not be called statically on objects in Eiffel, thus the appropriate translation of the `once` semantics is shown in listings 3.19:

Listing 3.19: A model for translating once using a lazy-loaded singleton.

```
interface Singleton {
  Singleton getInstance();
}
class Singleton$class implements Singleton {
  static class Singleton$class$OnceHolder {
    static Singleton instance = new Singleton$class();
  }
  public Singleton getInstance() {
    return Singleton$class$OnceHolder.instance;
  }
}
```

[25]See `http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom`.

This implementation relies on the well-specified initialization phase of execution within the JVM, which guarantees that instance would not be initialized until someone calls `getInstance()` method.

Subtleties of once

The `once` keyword introduces a few more subtleties as the semantics of the `once` keyword can be adjusted, thus allowing even finer grained control of once routine behavior. This is done by using `once` "keys", following the syntax `once("key")`, where valid once keys are "PROCESS", "THREAD" and "OBJECT". Table 3.4 (Meyer et al. [21, chap. 10]) shows how each of these keys affect `once` behavior:

`once` key	Routine executed the first time it is called ...
PROCESS	During process execution
THREAD	During each process thread execution
OBJECT	By each instance

Table 3.4: How once keys affect once routine execution.

If no key is specified, the default behavior is THREAD. In order to achieve this behavior on the JVM, an internal structure would be needed to contain such information during run-time. However, for this project, focus will be on the default behavior.

3.2.9 Assertions and Contracts

In Eiffel, there are six different kinds of assertions which can be used:

Checks: Plain assertion, where every boolean expression has to evaluate to `true`.

Precondition: Evaluated before entering the routine being executed. Preconditions can be inherited from parent features, and can only be weakened (e.g. only one of the preconditions has to hold).

Postconditions: Evaluated before leaving the routine being executed. Like preconditions, postconditions are inherited from parent features. But unlike preconditions, postconditions can only be strengthened meaning that all postconditions have to hold.

Class invariants: Evaluated between each feature execution. Every 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

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.

In Baumgartner [1, chap. 2, p. 39-43], a model on how to translate each assertion kind is described. The translation model follows the same pattern: Each assertion is a set of boolean expressions which has to evaluate to true at run-time, otherwise an AssertionError is thrown and the execution is halted at the position of the failed expression.

The proposed solution in this project, however, describes using a different method for translating assertions and contracts, by the use of the Java Modelling Language (JML). JML (Poll et al. [22] and Burdy et al. [23]) is a specification language for Java used to describe assertions and contracts following the DbC paradigm used in Eiffel, written as annotations in the `.class` files. Listings 3.20 shows an excerpt of an JML specification of the class `Purse` taken from Burdy et al. [23, p. 4], where the comments on the methods is the default syntax for defining contracts using JML:

Listing 3.20: A example of a Java class using JML annotation (from Burdy et al. [23]).

```
class Purse {
  /* Code omitted... */

  byte[] pin;
  /*@ invariant pin != null && pin.length == 4
    @       && (\forall int i; 0 <= i && i < 4;
    @                0 <= pin[i] && pin[i] <= 9);
    @*/

  /*@ requires   p != null && p.length >= 4;
    @ ensures    \result <==> (\forall int i; 0 <= i && i < 4;
    @                     pin[i] == p[i]);
    @*/
  boolean checkPin(byte[] p) {
    boolean res = true;
    for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
    return res;
  }

  /* Code omitted... */
}
```

Using JML instead of following the direct translatation approach, as described in Baumgartner [1], will allow the possibility to include a much more detailed description of the contracts specifications written in Eiffel when doing the translation to Java bytecode, as a variety of tools provide functionality based on JML annotations. The Iowa State JML tools is example of such, providing an assertion checking compiler *jmlc* which converts JML annotations into runtime assertions. It also includes a documentation generator `jmldoc`, which produces Javadoc documentation augmented with extra information from JML annotations, and a unit test generator *jmlunit* which generates JUnit test code from JML annotations. Other projects using the JML annotations includes functionality to do static analysis.

3.2.10 Exporting

Using the `export` statement, it is possible to either hide a feature of a parent class in a subclass, or export a hidden feature from a superclass in a subclass. An example is shown in listings 3.21:

Listing 3.21: Export statements in Eiffel.

```
class A
feature -- Visible to every body
  public_feature
    do
    end

feature {NONE} -- Private to A only
  private_feature
    do
    end

feature {B} -- Visible to B only
  feature_for_B
    do
    end

feature {A, B} -- Visible to both A and B
  feature_for_AB
    do
    end
end
```

As shown in listings 3.21, class `A` defines four different features: `public_feature`, `private_feature`, `feature_for_B` and `feature_for_AB`. The feature `public-_feature` is visible to all classes as there is no export explicity stated, meaning that it default exports to ANY, e.g. the ancestor of all classes in Eiffel. The feature private-_feature is private to the class A, is it defines the export statement to NONE. As for feature feature_for_B, this is exported to be used from class B only, where feature-_for_AB may be used from either class A or B.

In Java it is not possible to allow such fine-grained control over visibility, whereas visibility is restricted inheritance and package level. In Scala, however, the utility of visibility modifiers has been extended to what is found in Java as shown in listings 3.22, that includes the various ways qualified access modifiers works in Scala:

Listing 3.22: Class Foo and Bar.

```
package org.examples
class Foo {
    // Protected to class Foo and all subtypes of Foo
    protected[Foo] var john = 456
    private[Foo] var jane = 789

    // Protected to the package "org
    protected[org] def someTest = "test"

    // Protected to "this" instance only
    private[this] var bob = 789

    def m(f: Foo) = {
      var myFoo = new Foo
      var newBob = this.bob // OK

      // The following two statements are not allowed:
      if (f == this) this.bob = f.bob
      this.bob = f.bob
    }
}
class Bar {
  def m = {
    var myFoo = new Foo
    var myJohn = myFoo.john // Not allowed
    var someTestResult = myFoo.someTest // OK
  }
}
```

Class Foo declares the fields john and jane, which are restricted to the class Foo and subtypes of it. The method someTest is visible to all types in the package org, whereas the field bob is visible to this instance of the class Foo. Looking at the method m in

Foo, what is interesting is the fact that it is not possible to access bob at the parameter f of type Foo. This is due to the fact that the static checking is preventing this. Likewise, it is not possible to access the field john from Bar, since Bar is not a direct subtype of Foo. Given that Bar had been in another package than Foo, this would have yielded the same result as for john.

Listings 3.23 shows how the decompiled version of the compiled class Foo looks like:

Listing 3.23: Decompiled code for class Foo.

```
package org.examples;
import scala.ScalaObject;
public class Foo implements ScalaObject {

  private int john = 456;
  private int org$examples$Foo$$jane = 789;
  private int bob = 789;

  public int john()  { return this.john; }

  public void john_$eq(int paramInt) { this.john = paramInt; }

  public String someTest() { return "test"; }

  public int org$examples$Foo$$jane() {
    return this.org$examples$Foo$$jane;
  }

  public void org$examples$Foo$$jane_$eq(int paramInt) {
    this.org$examples$Foo$$jane = paramInt;
  }

  public void m(Foo f)  {
    Foo myFoo = new Foo();
  }
}
```

Considering the decompiled version of Foo some interesting points can be made:

protected[Foo] john: Is translated to a private field named john in the class with a public setter (`void john_$eq(int paramInt)`). Reasoning about the promises Scala makes about the visibility, this is clearly not reflecting their claim once compiled down to the JVM. Clients can freely, upon using the class inside of Java, change the value of john to their liking.

private[Foo] jane: Is translated to a private field jane with a public setter (`void org$-examples$Foo$$jane_$eq(int paramInt)`). It is noted that some name mangling occurs here. As in the first case, nothing prevents clients from actually changing the value of jane.

protected[Foo] someTest: Is translated to a public method. As in the two former cases, this means that clients can call someTest unrestricted.

private[this] bob: Is translated to a private field without any setters or getters.

From this example, one can conclude that Scala makes no restrictions on how methods and field may be called, once it has been compiled to the JVM. This means that all restrictions rules are enforced during static checking of the Scala syntax, and thus before the compilation to JVM bytecode takes place. The consequence is that it makes it possible for clients to call the compiled Scala code in some controversial ways that were not necessarily the intension. What is evident, is that Scala takes the simplest and fastest (performance wise) approach to solve this problem.

In order to enforce the restrictions, as would be the goal when translating the `export` statement in Eiffel to the JVM, one option is to make use of run-time checks to verify whether a class may access a given feature of a class. However, checking the caller of a method in Java during run-time is costly[26] to perform, so it would be of great burden to have to perform it at every method call. Therefore it may be arguable, that as long as the Eiffel compiler always check visibility rules statically, this will have make up for a viable solution. Hence it is decided to translate `export` statements found in Eiffel to the JVM as is done in Scala.

3.2.11 Expanded Types

Expanded types in Eiffel are *value types*, rather then reference types. A variable or class defined as `expanded` is such a value type. The two main properties of expanded types are:

1. An expanded type is never `Void`.
2. An expanded type is attached to its defining class, thus can not be shared among multiple objects.

[26]A discussion on this matter can be found at `http://tinyurl.com/294jtx7`

As described in Baumgartner [1, chap. 2, p. 35-36], the translation of an expanded type is the same as for non-expanded classes. The only differences are that assignment of value types should involve *cloning* of the object being assigned, rather referencing it, thereby getting a true copy of the object. This is something that the client of the translated code has to be aware of. Additionally, value types involved in a comparison for equality should compare the values of the objects and not the references.

3.2.12 Generics

An example on how to specify and use generic classes found in Eiffel is shown in listings 3.24:

Listing 3.24: An example of using generic classes in Eiffel.

```
class D [G]
feature
  g: G
    do
    end

  h (x: G)
    do
    end
end

class E [G -> D]
end
```

The notation `G -> D` shown in listings 3.24 means that `G` must be a descendant of `D`, which is also known as *constrained genericity*. With the support of generics in Java 5, the example shown in listings 3.24 can be translated directly as follows:

Listing 3.25: Translating generics in Eiffel to Java.

```
class D<G> {
  G g() { ... }
  h(G x) { ... }
}
class E<? extends D> { }
}
```

Here the type parameter `G` within the angle brackets of class `D` declares the generic type of the class. Moreover, since Java allows the use of type wildcards[27] to serve as type

[27]Wildcards are type arguments in the form "`?`", possibly with an upper or lower bound.

arguments for parameterized types. In the case of class E, one can use an upper bound of a type wildcard, where as the `extends` keyword is used, which indicates that the type argument is a descendant of the bounding class D. Moreover, had class E defined features that would have allowed retrieval of elements stored in some list inside of E, then an element could be retrieved and safely assigned to a D type, thus ensuring *covariance* which is also a construct available in Eiffel.

3.2.13 Rescue clauses

Exceptions in Eiffel occur if a contract is violated, if an assertion does not hold or an unexpected error is encountered. Such exceptions may be handled in a special part of a routine, namely a `rescue` block. If the code of the `rescue` block contains a `retry` instruction, the routine is re-executed, otherwise the routine fails and the exception is propagated to the caller of the routine. Listings 3.26 shows an example of a feature which contains a `rescue` block, where the routine code may be executed a maximum of ten times, after that the routine will fail:

Listing 3.26: An example of a `rescue` block in Eiffel.

```
call_risky_routine
  local
    retry_count: INTEGER
  do
    --| Risky code which may cause an unexcepted error.
  rescue
    --| Exception handling code.
    if retry_count < 10 then
      retry_count := retry_count + 1
      retry
    end
  end
```

In Java, the `rescue` block may be translated as shown in listings 3.27. A local variable `$retry` is generated, which controls the body of the feature. Next, the body of the feature is wrapped in a `try` and the `rescue` body is inserted in the `catch`:

Listing 3.27: Translation of `rescue`.

```
void call_risky_routine {
  boolean $retry = true; int retry_count = 0;
  while ($retry) {
    $retry = false;
```

```
      try {
        // Risky code which may cause an unexcepted error.
      } catch {
        // Exception handling code.
        if (retry_count < 10) {
          retry_count++;
          $retry = true;
        }
        if (!$retry) { throw; }
      }
    }
  }
```

3.2.14 Tuple Types

In Eiffel, tuples types may be viewed as a simple form of class, that only provides attributes and the corresponding "setter" procedure. An example of using tuple types in Eiffel is shown in listings 3.28:

Listing 3.28: An example of using once.

```
class REPOSITORY_FAKE
feature
  add_person (p: [name: STRING; age: INTEGER])
    do
      --| Save the person to the fake repository
    end
end
class A
feature
  repository: REPOSITORY_FAKE

  test_repository
    do
        --| Add a new person named John Doe of the age 60
        --| to the repository.
      repository.add_person (["John Doe", 60])
    end
end
```

In Baumgartner [1], a translation model for tuples types is not described, neither does the current release of the Java base library include[28] a tuple type which could have been used. Fortunately, from Java 5, generics can be used together with *type wildcards* to model a tuple type, which the Eiffel tuple type may then be mapped to during translation. An example of such an implemention can be found at `http://tinyurl.com/cdmzpvo`, which contains a simple type-safe tuple implementation.

3.2.15 Agents

In Eiffel, agents are threated as first-class citizens[29], representing operations as run-time objects. Agents can be handled as any other object: passed as parameters, assigned to variables and such. Additionally, agents may be inlined or created from an object's feature, and can contain both open and closed arguments.

In Java there is no direct equivalent, which is supported directly on the current version of the JVM. Other programming languages, such as Scala, implements a similar mechanism to agents, called *closures*. However, the work behind translation Scala closures onto the JVM is not trivial, in fact, it is quite hard (Pollak [24]) to implement in practice. Fortunately, JSR 335 (Oracle [25]), also known as *project lambda*, aims at implementing full support for the use of lambda expressions in the Java language, which is scheduled for Java 8. It is therefore decided to halt the implementation of this language feature until the next generation of the JVM is released.

[28]This was recently added to the .NET framework base library (.NET 4.0).

[29]See `http://en.wikipedia.org/wiki/First-class_citizen`

3.3 EiffelStudio - Under the Hood

Being able to extend the EiffelStudio compiler requires identification and understanding of the core components of the compiler that will be affected in process of adding a new target for code generation.

The following sections describes the analysis of identifying the important parts of the compiler. The analysis is based on different resources including: study- and reverse-engineering of the EiffelStudio compiler's source code, reading various online resources and attending online video conferences with the team at Eiffel Software. The first section gives an overview of the general structure and organization of EiffelStudio, whereas the rest of the sections focuses on the compilation process (using a top-down approach), the internal data structures created in this process as well as how the compiler can be extended.

Static and dynamic diagrams shown in the following sections adheres to the *Business Object Notation* (BON), as described in Waldén and Nerson [26][30].

3.3.1 Structure and Organization

At the source-level, EiffelStudio uses a set of *libraries*, *frameworks* and its own code. Figure 3.4 gives an overview on how the EiffelStudio is structured:

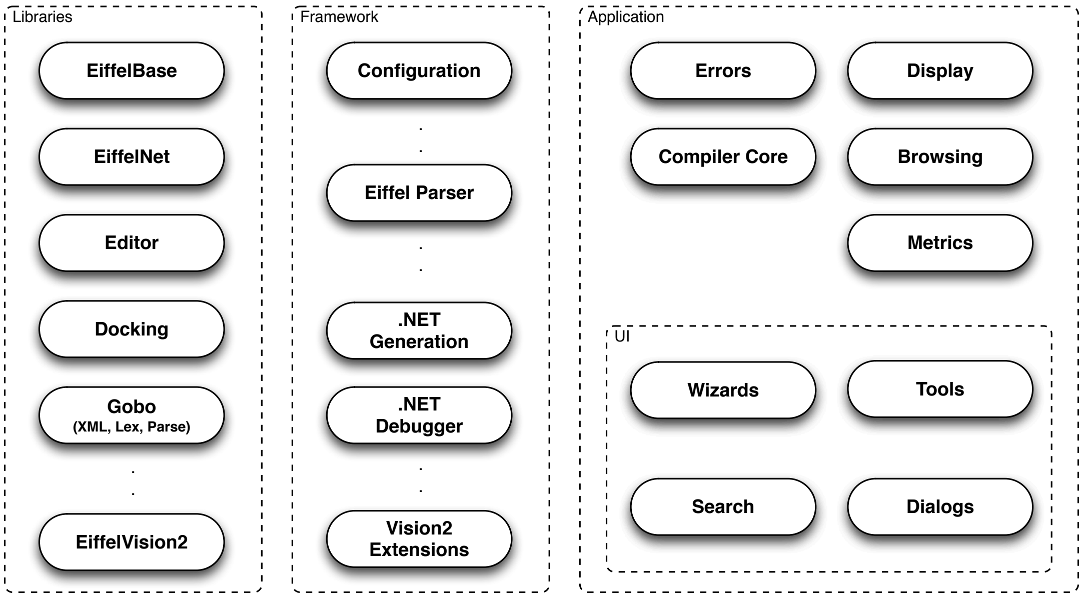

Figure 3.4: An overview of how EiffelStudio is structured and organized.

To the left are the different libraries which are reusable components used for general

[30]For a short introduction on the BON notation, please refer to Waldén [27].

purposes, both by EiffelStudio but also for external usage. In the middle are the frameworks, which are reusable components similar to the libraries, but specialized for EiffelStudio. Frameworks are not distributed as libraries, since these are mostly used internally within EiffelStudio. To the right are the components that represents the application code of EiffelStudio, which are divided into some major components: common, compiler and UI. In this project the components used have primarily been located in the *compiler core* component, along with a few libraries found in both *framework* and *libraries*.

3.3.2 The Complilation Process

This section describes how the compilation process in EiffelStudio is organized, as well as what is done in each stage during compilation. This is important in order to find out, where it is suitable to integrate the new language target compilation stage. The compilation process of an Eiffel project follows a *pipeline architecture* (Garlan and Shaw [28]), where the dataflow yields the source code transformation resulting from each stage (e.g from source code to an IR, then several annotated versions of the IR, and finally to a target language). Figure 3.5 gives a high-level overview of the steps involved in the compilation process:

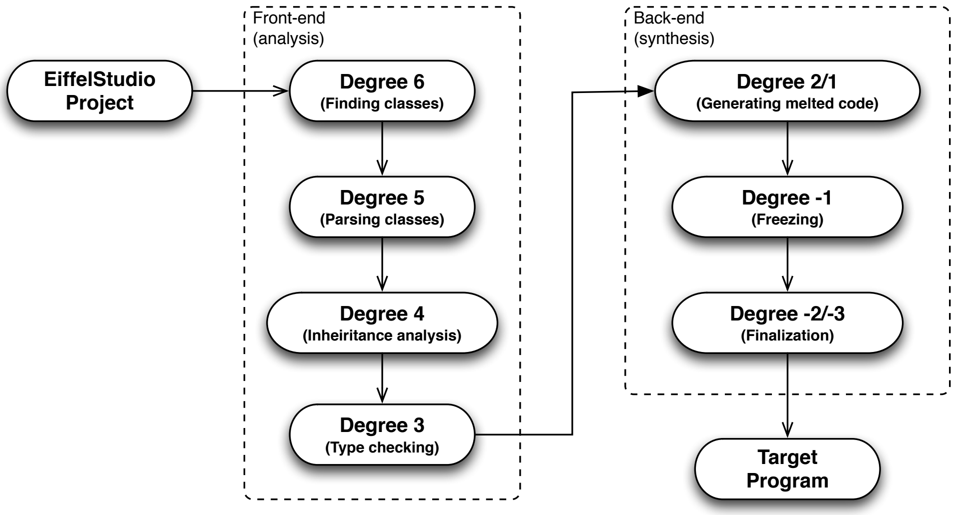
Front-end
(analysis)
Back-end
(synthesis)
EiffelStudio Project
Degree 6 (Finding classes)
Degree 5 (Parsing classes)
Degree 4 (Inheiritance analysis)
Degree 3 (Type checking)
Degree 2/1 (Generating melted code)
Degree -1 (Freezing)
Degree -2/-3 (Finalization)
Target Program

Figure 3.5: The compilation process of EiffelStudio.

The content (Software [29]) of each step involved in the different degrees of the compilation process can be further described as:

Degree 6 Using the configuration file (Ace file) for the project being compiled (e.g. its *.ecf file), the file system is traversed to find files with the *.e extension, and then lookup the class name associated with the *.e file.

Degree 5 Starting from the set of classes needed to be compiled (e.g. `ANY`, `STRING`, `INTEGER`, ...) and the root class of the system, each class is parsed, where each unparsed class encoutered in the parsed class will be added to the set of classes that needs to be compiled. For each parsed class, a `CLASS_AS` (that is an AST representing the class) node is created which is kept, and inheritance are initialized so that a topological sort can be done at the end of the degree. The topological sort ensures that all the parents of that class are located before that class in the topological sort. The AST is built using the gelex and geyacc from the Gobo distribution[31].

Degree 5 is complete once transitive closure of referenced classes is done, in other word when the set is empty.

Degree 4 Using the topological order, the inheritance clause of every class is analyzed to ensure they are valid, thus building their *feature table* using the `CLASS_AS` instance from degree 5. A feature table is basically a table where all available features for a class are registered. Within the feature table, a feature is indexed by either its *name or routine id*.

Degree 3 The next step is to validate the code of each routine, also using the `CLASS_AS` instance. When valid, a `BYTE_NODE` object is created that corresponds to a compiled version of the AST, which is simplified version of the original AST made for *code generation purposes*. Instead of having the actual feature text, the new AST have internal id's for that feature, which are used for the code generation.

Degree 2/1 This is where the melted code is generated.

Degree -1 This is where the target code is generated for the workbench, e.g. freezed.

Degree -2/-3 This is where the workbench code is optimized, e.g. finalized.

Based on the content of the different compilation steps it is unlikely that modifications will be needed in degree 6 to 3, as these are related to the analysis stages of the compiler. Hence, focus will be on degree 2 to -1, whereas the new JVM compilation stage should occur just after degree 3 finishes. In terms of finalization, e.g. degree -2 to -3, it is arguable that finalization may still needed in order to optimize the generated workbench code. However, such topic is out of the scope for this project.

[31]See `http://www.gobosoft.com/`

Internals of the compilation process

From a high-level understanding of the different steps involved in compiling an Eiffel project, the focus is shifted towards the perspective of the compilation process at the source-level. Here, focus will be centered around the classes involved in the different degrees of the compilation process, as well as how these relate. The intent is to improving the understanding of what parts of the compiler may need to be updated as a result of adding a new target language.

Figure 3.6 shows a simplified overview of the classes involved in the compilation process and how they relate. When a user in EiffelStudio requests compilation of a project, the class `EB_MELT_PROJECT_COMMAND` is used to start the compilation. In doing so, it instructs the associated Eiffel project (`E_PROJECT`) to be melted. When the project is instructed to be melted, it instructs the associated workbench (`WORKBENCH_I`) to compile every class that is found in the system, which in turn recompiles the configuration description of the project (using `LACE_I`).

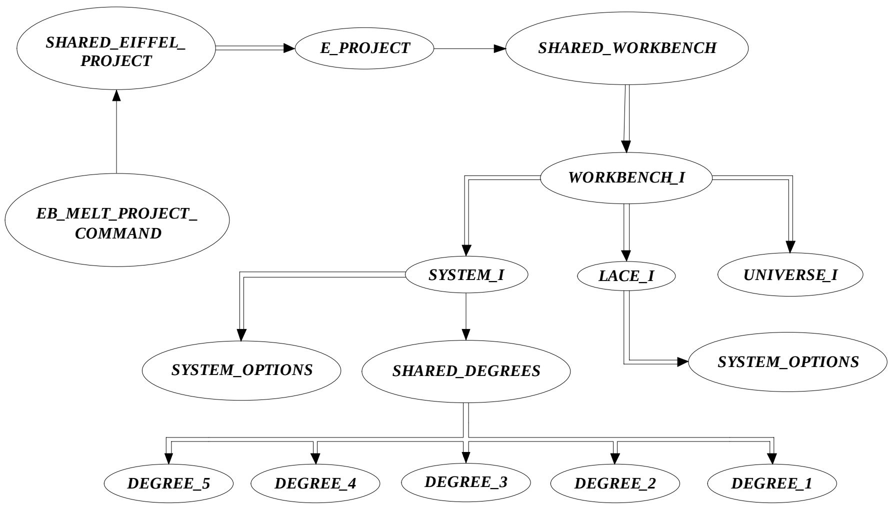

Figure 3.6: The classes involved in the compilation process.

During recompilation of the configuration description in `LACE_I`, a couple of events occur:

1. Retrieval of configuration description, which gets into an AST representing the structure of the specific configuration description.
2. The compilation target is computed using the AST.
3. The universe is build using the AST.

4. The internal settings is updated using the AST, e.g. the `SYSTEM_OPTIONS` class containing the options of the system, thus reflecting the options specified in the Ace file. The class `SYSTEM_OPTIONS` is used by the `SYSTEM_I` class during compilation to determine how it should proceed with compilation, once degree 3 finishes.

Once the configuration description has been recompiled, the workbench recompiles the classes in the system using the associated instance of `SYSTEM_I`. As a result, the different degrees from 5 down to -1 is executed and the system is recompiled.

From the perspective of adding a new language target, e.g. *"Java (JVM)"*, the part of the AST that represents the target of the configuration description should be updated, and accommodate for this when updating the internal settings. Currently, a configuration target already exists which is partially implemented, named `java_generation`. This should be used as a foundation for implemeting the new target. Moreover, `SYSTEM_I` should be updated to allow JVM bytecode generation to occur after degree 3 finishes, if a lookup in the internal settings indicates that the target of the project has been set "Java (JVM)".

3.3.3 Internal Data Structures

The following sections describes the various internal data structures created during the different compilation stages. This is to give an overview on the different parts used in the compiler, such as their purpose, how they interact and how they can be used. The descriptions of such will serve as a guideline for implementing the new back-end of the compiler, whereas knowledge on how to query the different parts of the internal data structure is needed.

An overview of the classes described in these section, and how these are related, is shown on figure 3.7 on page 51.

Classes

During degree 5, every parsed class gets an instance of the class `CLASS_I`. `CLASS_I` stores information about the file holding the class text, such as modification date, class name, associated cluster and so on. This class is the uncompiled representation of a class, which means that every class in the eiffel *universe* has a `CLASS_I` instance.

Compiled classes, e.g. classes in the eiffel *system*, also has an associated instance of the class `CLASS_C`. The `CLASS_C` is what the compiler is using internally to extract

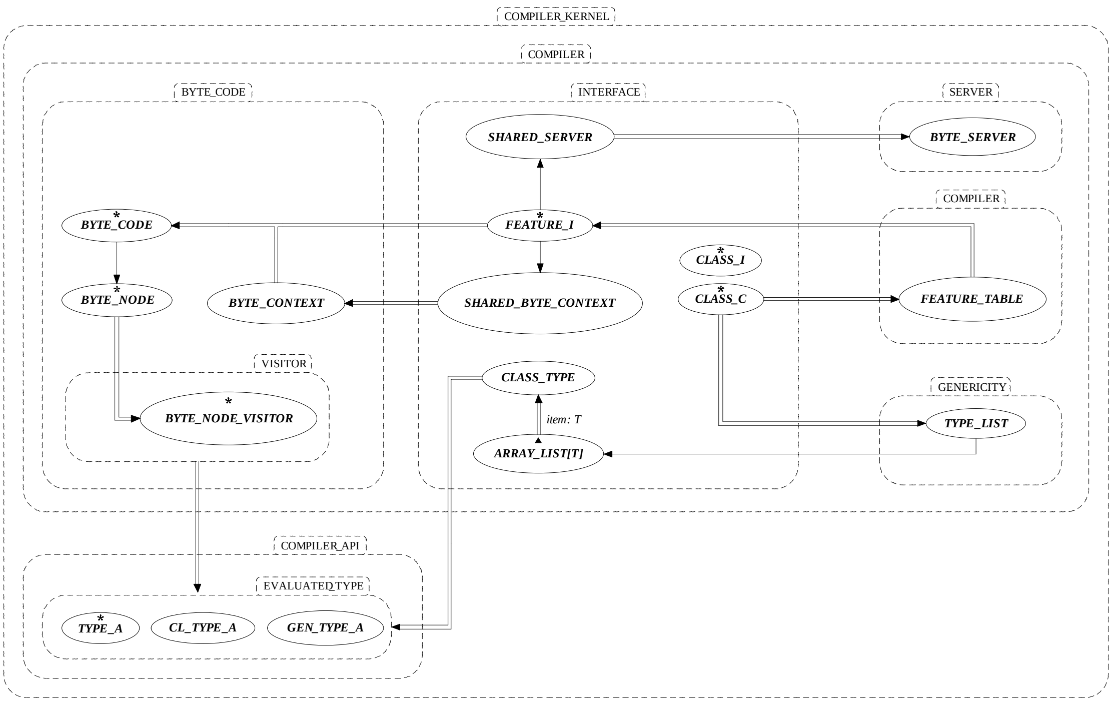

Figure 3.7: The internals of the compiler.

information about the class, whereas CLASS_I is mostly used at degree 6 to associate names to a file name. CLASS_C it is not the AST, but some specific information about its relations between classes, e.g. its ancestors, descendants, clients and suppliers as well as its features along with more information.

In the Eiffel terminology, one works with the notion of classes and types, where CLASS_C holds information about its types. This information is contained in a list of class types which is an instance of class TYPE_LIST, which is a container for instances of class CLASS_TYPE. For non-generic class, there is one possible type, for a generic class, there are as many types as there are actual generic derivations of a class (i.e. LIST[G] is the class, and LIST [INTEGER], LIST [STRING], LIST [ANY] are 3 types). When talking about types, one may refer to instances of class TYPE_A (and descendants such as CL_TYPE_A, GEN_TYPE_A, . . .). The CLASS_TYPE class contains information its actual type which can be either CL_TYPE_A, which is used for normal non-generic classes, like STRING, or GEN_TYPE_A which is used for generic classes, like LINKED_LIST of INTEGER.

When generating interfaces and class implementations, this information will be needed as it allows the new back-end to determine the appearance of the class being generated, as well as defining whether the class should be a generic or non-generic.

Features

Every CLASS_C stores the features of a class. The features of a class are stored in CLASS_C into an instance of FEATURE_TABLE, which is a container for instances of FEATURE_I. A feature is represented by an instance of class FEATURE_I, which has many descendats (about 30-40) including:

- PROCEDURE_I: Procedure call
- DYN_FUN_I: Dynamic function call, or simply a function call
- ATTRIBUTE_I: Attribute call
- EXTERNAL_I: External call

Depending on what concrete call is made, the appropriate descendant of FEATURE_I is called. For instance, making a call to a procedure would render the feature as an instance of PROCEDURE_I. FEATURE_I holds information about its final name, the class name where it is written, the specific type of the feature (e.g. an attribute, a procedure, a function, etc.), whether the feature is deferred and so on. Furthermore, for each FEATURE_I exists queries such as to_melt_in and to_generate_in. They can help figuring out when to generate a routine or not. It also contains the body id of its content, which is used to retrieve the actual bytecode from what is called the BYTE_SERVER, which essentially is a server that *serves* byte code for routines, indexed by the body index. In order to get the byte code of a feature, one has to retrieve this from the BYTE_SERVER, which is then loaded into an object of type BYTE_CONTEXT, which is a context used for code generation that encapsulates all that is related to the instructions of a feature body. Once the byte code has been retrieved into the context, an object of class BYTE_CODE, which is an descendant of class BYTE_NODE, is assigned to the context which represents the internal structure of the byte code for a routine. The byte code context then initializes the BYTE_CODE structure.

The BYTE_CODE class contains information about the arguments of the feature, its return type, the list of local variables defined, rescue clauses, any contracts defined and a list of BYTE_NODE instructions found inside of the feature, called its *compound*. The class BYTE_NODE represents the base class for all bytecode objects, which may be either an instruction, an expression, a binary operators and so forth. Every variant of the BYTE_NODE class exists under the subcluster named BYTE_CODE, as seen on figure 3.7 on page 51.

3.3.4 Extension Points

With an understanding of the classes involved in the compilation process, and how the various internal data structures can queried and used for generating the interfaces and class implementations, the last piece of information needed is how the compiler can be extended with regards to generating the content of the byte nodes. To process and work with the byte code of a feature from inside of the compiler, the current design rests on a *visitor* (Gamma et al. [18, p. 331-344]) class `BYTE_NODE_VISITOR`. Figure 3.8 focuses on the `BYTE_CODE` cluster, in which the different representations of each byte node lives along with the visitors used to process them.

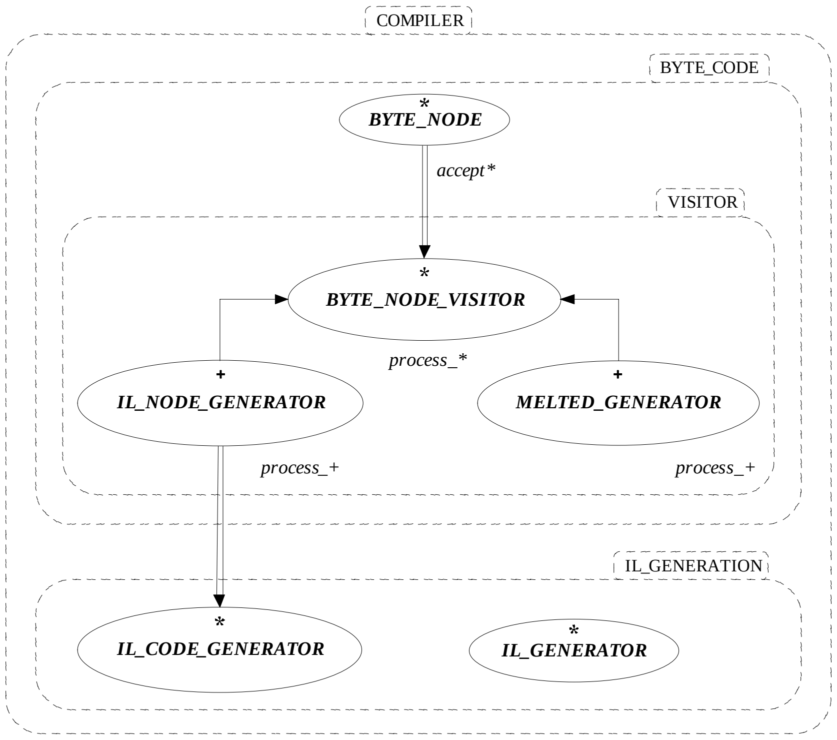

Figure 3.8: The extension points which can be used as hooks for implementing a new JVM code generator.

Leveraging this design, one option is to create a new visitor class, say `JVM_GENERATOR`, inheriting from `BYTE_NODE_VISITOR` used process each byte node and generate JVM bytecode. Doing so will involve mimicking what is done in the other visitor classes. However, when comparing the bytecode structure of .NET and the JVM (Sestoft [30, chap. 9, p. 183]), it is seen that these share a substantial amount of similarities. As such, it may be feasible to use reuse the existing `IL_NODE_GENERATOR` visitor class, along with the classes `IL_CODE_GENERATOR` and `IL_GENERATOR` (which are the classes used for generating the interfaces and class implementations, and the .NET bytecodes for each feature on a class when using the .NET code generation), in order to avoid some code duplication. Unfortunately does further analysis of the .NET visitor class show that many core .NET specifics details are embedded directly into the visitor. To use this class would require a refactoring, where all the .NET specific details where stripped out, thus having a generic IL visitor which could be used by either the .NET CLR, the JVM or any VM sharing a similar structure to those. Not doing so would result in having a class that violates the principle of having strong cohesion, as it would merge the specific implementations details for the .NET with the details of the JVM in one class. During a consultation with the Eiffel Software team, it was adviced to start off by using the .NET visitor. If this approach worked, then the team would help refactoring the visitor class to be generic, thus stripping it from the embedded .NET details. If the visitor class introduced too many problems, the fallback solution should be to create a new visitor class for the use in generating JVM bytecode.

Chapter 4

Implementation

4.1 The Eiffel Java bytecode Library (EJBCL)

This section presents the implementation of the Eiffel Java bytecode Library (EJBCL), which is used to generate `.class` files and its content. The overall architecture of the EJBCL is shown in figure 4.1. As the output model of the EJBCL library may be refered to, as a meta-model for the `.class` file format, consequently the architecture follows is an *object-oriented meta-model architecture*.

In essence, the library is composed of three clusters; `CLASS_FILE`, `CODE_GENERATION` and `COMMON`. The `CLASS_FILE` cluster contains the classes needed to describe the output model for the `.class` file format, the `CODE_GENERATION` cluster contains classes needed for generating Jasmin assembly code files, and the `COMMON` cluster contains miscellaneous shared information and utility classes used by the library. Examples on how to use the EJBCL can be found in Appendix C.

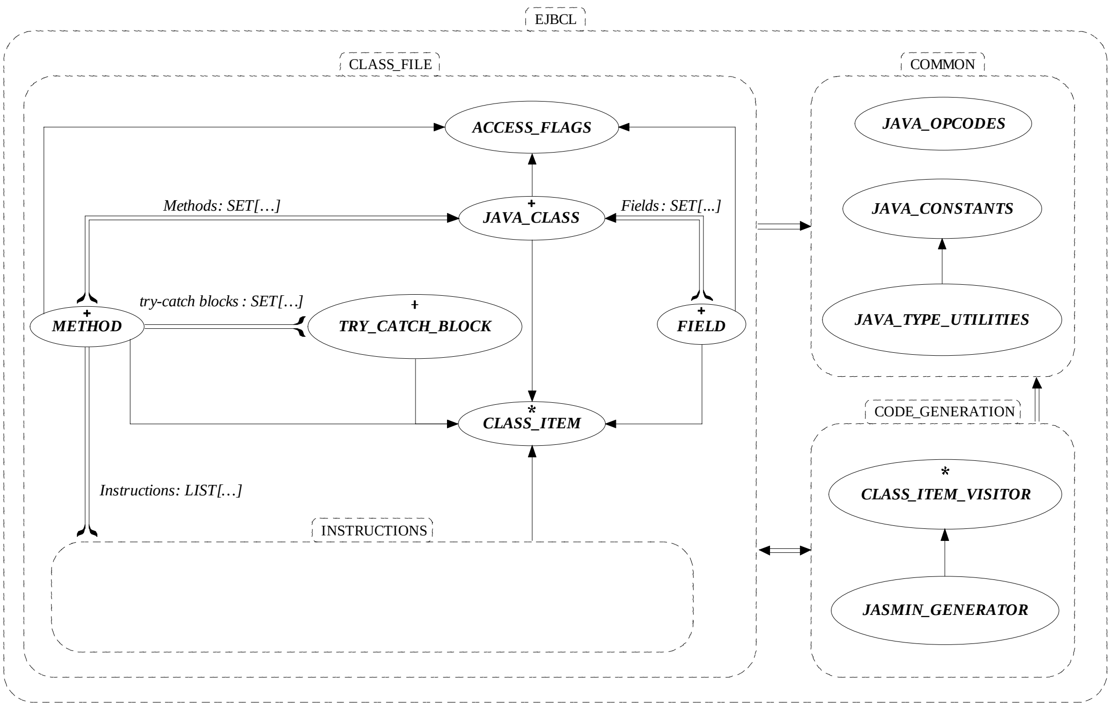

Figure 4.1: The overall architecture of the EJBCL, the content of the cluster `INSTRUCTIONS` is shown on figure 4.2 on page 56.

In the following, focus will be on the clusters `CLASS_FILE` and `CODE_GENERATION`, the classes they include and how these interact, as these holds the most important parts of the library.

4.1.1 The Output Model

Looking at the cluster CLASS_FILE shows that it contains the classes needed to the describe the different parts, or *items*, of the output model. An item is modelled using the deferred class CLASS_ITEM and its descendants.

The JAVA_CLASS class depicts the general notion of a .class file, including the header area, the field area and the method area. The header area is encoded using attributes, where each attribute corresponds to what is expected to be found in the header area, like the name of the class, its super class and implemented interfaces. The field area is represented as a list of FIELD objects, where a field has the attributes of its parent class, a name, a type descriptor and perhaps a value depending on whether the field is final or not. The method area is represented as a list of METHOD objects, where a method has the attributes of its parent class, a name, a type descriptor, a list of exceptions it may throw, a list of try-catch blocks, a list of instructions and information about the maximimum locals- and stack height.

The list of instructions within a method contains objects that are descendants of the deferred class ABSTRACT_INSTRUCTION[OPCODE->INTEGER, ARGS->TUPLE] located in the subcluster INSTRUCTIONS (see figure 4.2). This subcluster contains classes corresponding to the different JVM bytecode instructions found in Lindholm et al. [6, chap. 6], as described in the following.

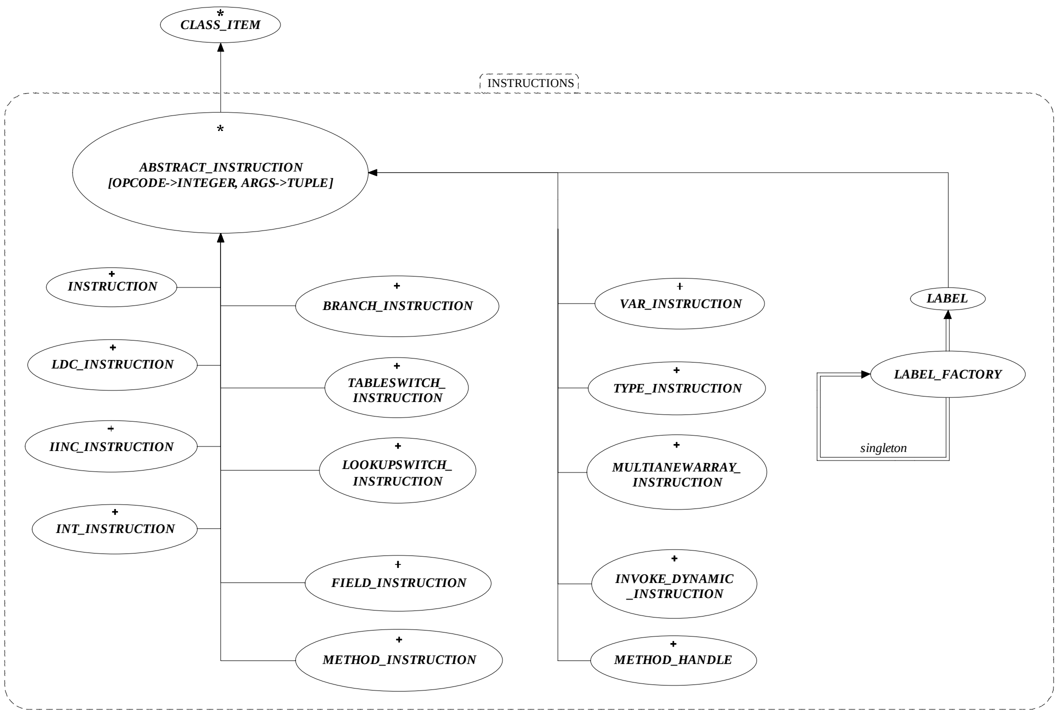

Figure 4.2: The instruction set.

The Instruction Set

A bytecode instruction is modelled using the deferred class `ABSTRACT_INSTRUCTION [OPCODE->INTEGER, ARGS->TUPLE]`, which is the *base class* for all bytecode instructions contained in a method's body. Every descendant of the instruction base class represents one or more instructions that shares the same set of arguments. Moreover, where possible, instructions are grouped after family, meaning that the `METHOD_INSTRUCTION` handles all that is related to *method invocation* and so on.

The instruction base class contains two different creation routines to be used by its descendants, one for creating an instruction *with no arguments*, and one for creation an instruction *with one or more arguments*. Both creation routine takes in the textual representation of the opcode, the actual opcode integer as well as the number of stack slots that the operand stack height will change when the instruction is executed.

The two generic parameters represents the mnemonic JVM opcode that identifies the instruction along with a fixed number of arguments required by that specific opcode. An important note is the use of the `TUPLE` type for the arguments of an instruction. Alone among all classes, the `TUPLE` class has a variable number of generic parameters. `TUPLE`, `TUPLE [X]`, `TUPLE [X, Y]`, `TUPLE [X, Y, Z]` and so on are all valid types, assuming that `X`, `Y`, `Z` are valid types. The conformance rule[32] of the `TUPLE` is as follows:

Conformance Rule: *For $n >= 0$, `TUPLE [`U_1`, `U_2`, ..., `U_n`, `$U_n + 1$`]` conforms to `TUPLE [`U_1`, `U_2`, ..., `U_n`]` and hence to `TUPLE [`T_1`, `T_2`, ..., `T_n`]`, if each of the U_i conforms to each of the T_i for $1 <= i <= n$. In particular all tuple types conform to `TUPLE`, with no parameter.*

When applied to the design of the instruction set, this rule shows that descendants of the instruction base class can specify their own variant of the `ARGS->TUPLE` generic parameter in terms of having a unique set of generic parameters corresponding to the arguments required by the opcode of the instruction. An example of such can be seen in the code listings in listings 4.1 on page 58. This approach is quite effective, as it allows the `METHOD` object to define a list of instruction base class objects, whereas the details of the arguments are hidden. Additionally, as seen in the following section on code generation, the arguments of each instruction are fully type-safe, meaning that it is possible to refer directly to the invividual arguments by name because of the built-in functionalities of the `TUPLE` type.

[32]See `http://archive.eiffel.com/doc/manuals/language/tuples/page.html` for the full description.

Listing 4.1: Example of an instruction with arguments.

```
class METHOD_INSTRUCTION
inherit
  ABSTRACT_INSTRUCTION [INTEGER,
              TUPLE [type: STRING; name: STRING; desc: STRING]]
    -- Code omitted...
end -- class METHOD_INSTRUCTION
```

To ensure that instructions are created using only valid opcodes, where each descendant instruction provides an general factory method, called `make`, which takes an opcode as one of its arguments. Depending on the opcode supplied by the client, an appropriate type of the instruction is created. For every opcode, an additional set of features are defined in the instruction class, encapsulating the creation logic of each type of instruction. These features are hidden to the clients of the instruction, thus only visible to the instruction class. An instruction is only created under the constraint that the opcode provided is *valid*, and supported by the given instruction's factory method, where a valid opcode means that the opcode provided has to be in compliance with the opcodes defined in Lindholm et al. [6, chap. 6]. To ensure the constraint of using valid opcodes is met, the `JAVA_OPCODES` class found in the `COMMON` cluster contains all the available opcodes found on the JVM, where each opcode is represented as an `INTEGER` constant listed by the name of the opcode with value corresponding to the hexidecimal value of the opcode. If the client provides either an opcode that is not supported by the instruction, or an opcode not valid in general, an assertion error is thrown during run-time upon creating the instruction object.

Encoding the `.class` specifications into the Output Model

In the description of the `.class` file format in Lindholm et al. [6, chap. 4], the `.class` file, its fields and methods are each bound to a set of constraints, some refering to particular sections of the Java Language Specification (JLS), as described in Gosling et al. [31]. To embed these constraints into the design of the library, the semantics of each constraint has been encoded as *class invariants* using first-order logic. For example, consider the constraint *JLS §9.1.1.1* as described below:

If the `ACC_INTERFACE` flag of this class file is set, its `ACC_ABSTRACT` flag must also be set. Such a class file must not have its `ACC_FINAL`, `ACC_SUPER` or `ACC_ENUM` flags set.

The semantics of the constraint tranlates into the equivalent logical expression:

$$interface \Rightarrow (abstract \land \neg(final \lor super \lor enum))$$

Listings 4.2 shows the complete set of class invariants for the class `JAVA_CLASS`, which conforms to the semantics of the contraints found in the general `.class` file, including JLS §9.1.1.1.

Listing 4.2: The encoded specifications for a general class.

```
class JAVA_CLASS

  -- Code omitted...

invariant
    -- JVMS: Table 4.1. Class access and property flags:
  valid_access_flags: is_public or
            is_final or
            is_super or
            is_interface or
            is_abstract or
            is_synthetic or
            is_annotation or
            is_enum

    -- JLS §9.1.1.1:
  jls_9_1_1_1: is_interface implies (is_abstract and not (is_final or
      is_super or is_enum))

    -- JLS §8.1.1.2:
  jls_8_1_1_2: (is_annotation implies is_interface) or
        (not is_interface implies (not is_annotation and (not is_final or
            not is_abstract)))

end -- class JAVA_CLASS
```

The different access modifers used to define the class invariants originates from the class `ACCESS_FLAGS`, which the classes `JAVA_CLASS`, `METHOD` and `FIELD` all inheirit from (see figure 4.1 on page 55). Having the constraints encoded as contracts means that violations to library, and thereby the `.class` file format is disallowed. As such, when using the EJBCL clients are constrained in the sense that they will have to follow the rules specified for the `.class` file format. This renderes the library robust in the sense that clients can only do what is allowed according to the specification of the JVM.

4.1.2 Generating Code

Once the output model has been created, the next step is to traverse the output model and generate the Jasmin assembly code. To do so, the design makes use of a visitor[33] class to perform to code generation operations on each of the elements of the output model structure.

Refering to the deferred CLASS_ITEM class from listings 4.1, it has been augmented with an accept feature to let it work with a visitor:

Listing 4.3: An excerpt of the CLASS_ITEM class.

```
deferred class CLASS_ITEM
  -- Code omitted...

feature -- Basic operations

  accept (a_visitor: CLASS_ITEM_VISITOR)
      -- Accept operation.
    require
      not_void: a_visitor /= Void
    deferred
    end
end -- class CLASS_ITEM
```

When the CLASS_ITEM class is effected by its descendants, the effected class will contain different attributes, such as their name, a list of implemented interfaces and type descriptors. The deferred class for all visitors of CLASS_ITEM has deferred features for each descendant of CLASS_ITEM, as shown next:

Listing 4.4: An excerpt of the CLASS_ITEM_VISITOR class.

```
deferred class CLASS_ITEM_VISITOR
  -- Code omitted...

feature -- Basic operations

  process_java_class (a_java_class: JAVA_CLASS)
      -- Visit a 'JAVA_CLASS' item.
    require
      a_java_class_not_void: a_java_class /= Void
```

[33]The purpose of using a visitor to generate the assembly language was chosen, as it allows further extensions of the library in case that one might emit a different sort of assembly like, do pretty-printing or so, without changing the structure of the output model.

```
    deferred
    end

  process_field (a_field: FIELD)
      -- Visit a 'FIELD' item.
    require
      a_field_not_void: a_field /= Void
    deferred
    end

  process_method (a_method: METHOD)
      -- Visit a 'METHOD' item.
    require
      a_method_not_void: a_method /= Void
    deferred
    end

  process_method_insn (a_method_insn: METHOD_INSTRUCTION)
        -- Visit a 'METHOD_INSTRUCTION' item.
    require
      a_method_insn_not_void: a_method_insn /= Void
    deferred
    end

  -- Code omitted...

end -- class CLASS_ITEM_VISITOR
```

CLASS_ITEM descendants define accept in basically the same way: It calls the CLASS_ITEM_VISITOR feature that corresponds to the class that received the accept request, like this:

Listing 4.5: The accept of the JAVA_CLASS class.

```
accept (a_visitor: CLASS_ITEM_VISITOR)
    -- <Precursor>
  do
      -- Process the class.
    a_visitor.process_java_class (Current)
  end
```

CLASS_ITEM that contains other CLASS_ITEM implements accept by iterating over its children and calling accept on each of them. This is indeed the case for the JAVA_CLASS class which has lists of both fields and methods (which are descendants of CLASS_ITEM).

Next, the visitor needed for generating Jasmin assembly file code is defined:

Listing 4.6: An excerpt of the `JASMIN_ASSEMBLY_GENERATOR` class.

```
class JASMIN_ASSEMBLY_GENERATOR
inherit
  CLASS_ITEM_VISITOR
    -- Code omitted...

feature -- Basic operations

  process_method_insn (a_method_insn: METHOD_INSTRUCTION)
        -- Visit a ‘METHOD_INSTRUCTION’ item.
      do

        if a_method_insn.opcode.is_equal ({JAVA_OPCODES}.invokeinterface)
        then
          write_line (a_method_insn.textual_name + " " + a_method_insn.
              arguments.type + "/" +
              a_method_insn.arguments.name + a_method_insn.arguments.desc
                  + " " +
              (create {JAVA_TYPE_UTILITIES}).get_argument_size (
                  a_method_insn.arguments.desc).out)
        else
          write_line (a_method_insn.textual_name + " " + a_method_insn.
              arguments.type + "/" +
                a_method_insn.arguments.name + a_method_insn.arguments.
                    desc)
        end
      end

  -- Code omitted...

end -- class JASMIN_ASSEMBLY_GENERATOR
```

Among other, the `JASMIN_ASSEMBLY_GENERATOR` will generate Jasmin assembly file code for any `METHOD_INSTRUCTION` bytecodes found in the list of instructions of a `METHOD` object. The full implementation of the `JASMIN_ASSEMBLY_GENERATOR` class contains features for generating code for each part of the output model structure.

As a finalizing step, the Jasmin assembler is used to generate `.class` files from the Jasmin assembly code files generated by the visitor. Please refer to Appendix C for a sequence diagram of how the code is generated.

4.2 Extending the EiffelStudio Compiler

This section presents the implementation of the JVM back-end. This section is divided into two sub-sections; *preparation* and *generation of* `.class` *files*.

The first section will address the issues of setting up the development environment, as a test project will have to be setup correctly in order to use it when developing on the back-end of the compiler. Once the development environment has been configured, the second section will focus on the implementation of the JVM back-end.

4.2.1 Preparations

Before starting the development on the JVM back-end, there are two tasks that must be done first:

- The development environment must be set up correctly.
- A test project must be created, which does not use any pre-compiled libraries.

In setting up the environment, latest version[34] of the EiffelStudio IDE should be installed, as the source code of EiffelStudio will require the lastest version of the compiler in order to compile.

Next, the Eiffel Verification Environment (EVE) branch must be configured. EVE is a project that runs parallel to the commercial branch of EiffelStudio, whereas the goal of EVE is to unify all the efforts that aim at adding new, experimental functionalities to EiffelStudio. Instructions on how to reach and setup the EVE branch may be found at the EVE wiki[36], which contains guides for both the Windows and UNIX platform.

Once EVE have been set up, a test project must be created. The purpose of having a test project is that it enables conducting small tests when implementing the JVM back-end, such as writing a class with a limited number of feature constructs and such. From within the released version EiffelStudio, a new project must be created, e.g. *project_test*, which contains no more then a root class. When the project has been created, the configuration file of the project should be located and opened. The content of an Eiffel project configuration file will look somewhat to what is shown in listings 4.7 on page 64.

[34]Which can be found at either Eiffel Software website [35] for Windows users, or at MacPorts for Mac OS users.

[36]See `https://trac.inf.ethz.ch/trac/meyer/eve/wiki`

Listing 4.7: An excerpt of the Eiffel project configuration file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<system ... name="project_test">
  <target name="project_test">
    <root feature="make" class="APPLICATION"/>
    <option warning="true">
      <assertions precondition="true" postcondition="true" check="true"
          invariant="true" loop="true" supplier_precondition="true"/>
    </option>
    <precompile name="base_pre" location="$ISE_PRECOMP/base.ecf"/>
    <library name="base" location="$ISE_LIBRARY/library/base/base.ecf"/>
    <cluster name="project_test" location=".\" recursive="true">
      <file_rule>
        <exclude>/EIFGENs$</exclude>
        <exclude>/.svn$</exclude>
        <exclude>/CVS$</exclude>
      </file_rule>
    </cluster>
  </target>
</system>
```

In order to prevent EiffelStudio from using any pre-compiled libraries while developing the new back-end, the line `<precompile .../>` *must* be removed. Additionally, in order to instruct the compiler to use the `java_generation` target, the following code must be added to the project configuration file, after the first `<target>...</target>` block:

Listing 4.8: The code needed to compile the project using the JVM as target.

```
<target name="project_test_jvm" extends="project_test">
  <setting name="java_generation" value="true"/>
</target>
```

4.2.2 Generating `.class` files

In order to generate JVM bytecode, a new subcluster in `COMPILER_KERNEL` (see figure 3.7 on page 51) has been added, named `JVM_GENERATION`, which contains the required classes for processing Eiffel class files into `.class` files. In addtion, the EJBCL has also been added, as shown in figure 4.3 on page 65.

In generating `.class` files from the Eiffel source code, the compiler must be instructed to output JVM bytecode, which requires a modification of the class `SYSTEM_I`. More precisely, when degree 3 finishes in the recompilation phase, SYSTEM_I must be notified to start the generation of JVM bytecode, if the java_generation flag is set to true. To do so, a new routine is introduced in SYSTEM_I, named generate_jvm, which leverages on the class JVM_GENERATOR.

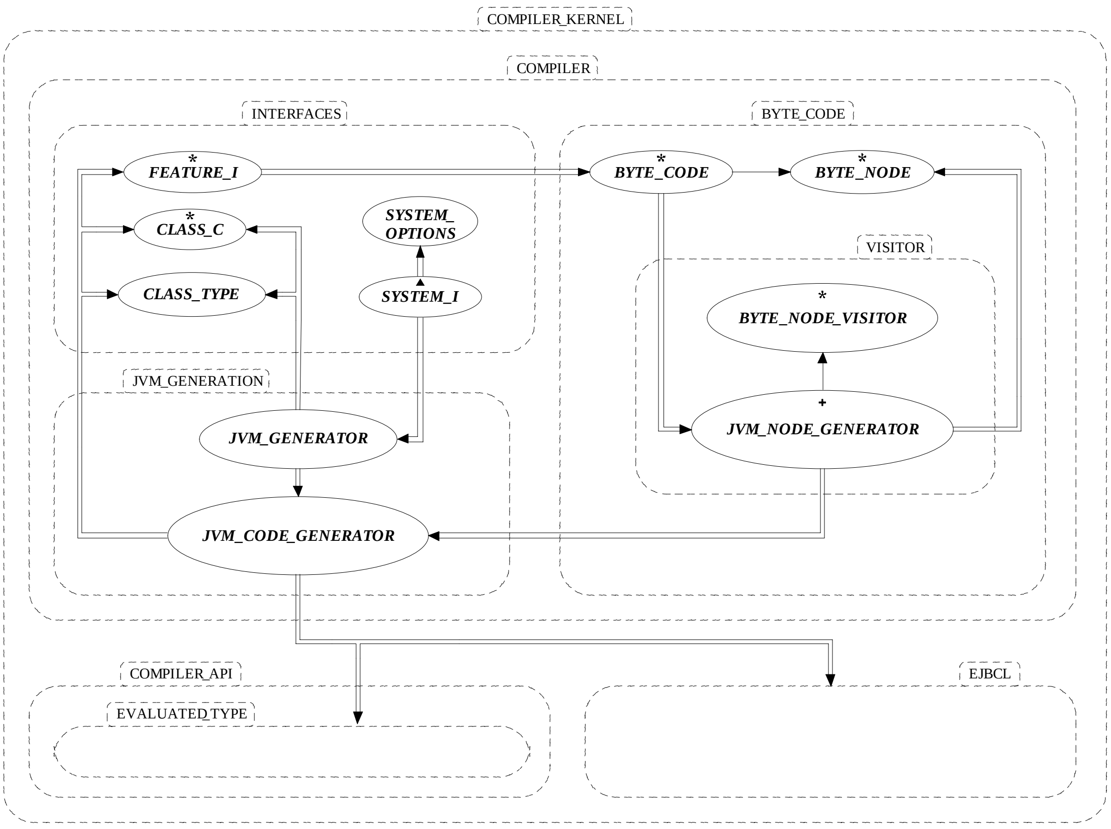

Figure 4.3: An overview of the new back-end classes added to the compiler.

The JVM_GENERATOR class is the main entry-point used for generating .class files, which contains two general routines: generate and deploy, which are both called from the generate_jvm routine in SYSTEM_I in that order. The generate routine is responsible for generating and assembling the generated Jasmin assembly code files produced by the JVM_CODE_GENERATOR class into .class files, while deploy is responsible for packing the .class files into JARs, moving these into the correct output folder and finalizing the compilation by cleaning up the temporary produced files. The conceptual understanding of the different stages occuring in the code generation, are shown in figure 4.4 on page 66.

Generating Interfaces and Class Implementations

Before generating the interfaces and class implementations, the system classes must be sorted topologically. The topological sort will ensure that all the parents of a class are

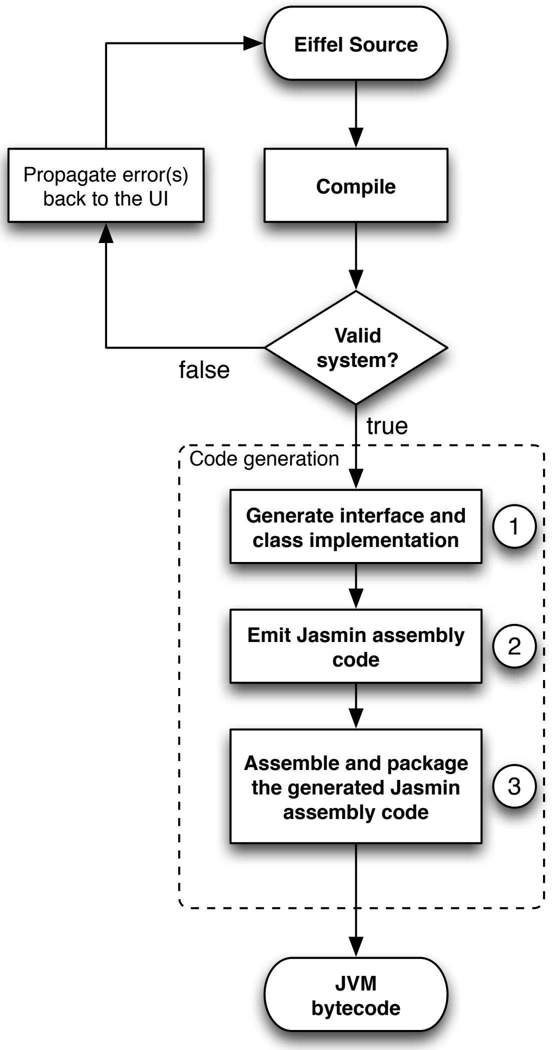
Eiffel Source
Compile
Propagate error(s) back to the UI
Valid system?
false
true
Code generation
Generate interface and class implementation
1
Emit Jasmin assembly code
2
Assemble and package the generated Jasmin assembly code
3
JVM bytecode

Figure 4.4: Flow-chart diagram of the JVM bytecode compilation process

located before that class in the topological sort.

Once the system classes is sorted, the next step is to generate JVM bytecode for each of these classes. This yields traversing the list of system classes, and then traversing each type of that class. Then for each type of that class, an interface and corresponding class implementation is created.

To generate the class interfaces and class implementations, the class `JVM_CODE_GENERATOR` is used, which exposes two features: `generate_interface` and `generate_class-_implementation`, which occurs in stage 1 as shown in figure 4.4.

The algorithm for generating interfaces used by `generate_interface` is shown in algorithm 1 on page 67.

Algorithm 1 Algorithm for generating interface classes

```
procedure generateInterface($c, ct$)    ▷ Generate the interface based on the class $c$ and class type $ct$
    Generate a new interface class using the EJBCL using $c$ and $t$
    for all $c.parents$ do
        Add the name of the parent class to the list of interfaces for the interface class
    end for
    for all $c.features$ do
        if the feature is not private by the class (e.g. exported to NONE) then
            Create a new method object using the information from the feature
            Add the new method object to the list of methods for the interface class
        end if
    end for
    Add the new interface class to the repository of generated Java classes
end procedure
```

Moreover, the algorithm for generating class implementations used by `generate_class-_implementation` is shown in algorithm 2 on page 68.

When every class in the system has been generated, the `JVM_CODE_GENERATOR` class will contain a list consisting of generated Java classes and interfaces, which is represented by the internal output structure of the EJBCL (e.g. `JAVA_CLASS`, `METHOD`, `FIELD`, . . .). To generate the Jasmin assembly code files, the list of generated Java classes and interfaces are passed a visitor of type `JASMIN_GENERATOR` which then traverses the output structure, thus generates the Jasmin assembly code files. Once the process is done, the Jasmin assembler should be called to assemble the `.class` files, where class files should be organized into folders corresponding to their cluster name(s) (see stage 2 on figure 4.4 on page 66).

Once the `.class` files are created and organized into their respective cluster folders, the final step is to package[37] these into JAR files, as well as moving these to the correct output folder (see stage 3 on figure 4.4 on page 66).

[37]For more information on how to generate JAR files, please refer to `http://docs.oracle.com/javase/tutorial/deployment/jar/build.html`

Algorithm 2 Algorithm for generating class implementations

```
procedure generateImplementation($c, ct$) ▷ Generate the class impl. based on the class $c$ and class type $ct$
    A Java class is generated using the information of the class type, where the name is postfixed with $class
    for all $c.parents$ do
        Add the name of the parent class to the list interfaces for the interface class.
        Generate a field to hold a reference of the delegate object. The name of the field should follow the naming convention <class-name>$delegate. Each of these fields must then be added to the list of fields for the Java class.
    end for
    Create and add a default constructor, which is marked as being protected.
    for all $c.creations$ do
        Create and add a new factory method corresponding to each creation feature to. Each factory method will create an instance of the class using the generated constructor, thus set the value of the new object as specified in the creation feature, thus return it.
    end for
    for all $c.features$ do
        Create a new method object using the information from the feature
        if the feature is deferred then
            Mark the new method object as protected abstract. Moreover, mark the class implementation as abstract as well.
        else if the feature is new, either effected or redefined then
            Instruct the byte node AST visitor to start generating the code of the feature's body
        else
            The implementation of one of the parents features of the current feature is called through one of the delegation objects
        end if
        Add the generated method to the list of methods of the class implementation
    end for
    If the class is marked as the root class, then a main method is generated and added to the class implementation, which calls the creation method make of the root class
    Add the generated class implementation to the repository of generated Java classes
end procedure
```

Chapter 5

Evalulation and Verification

5.1 Evaluate and verify the generated JVM bytecode

When evaluating and verifying the generated JVM bytecode, there are two distinct points to be made, which deserves a brief discussion.

In order to verify that the existing, internal structure of the compiler has not been broken, the EiffelStudio compiler source code contains various contracts to assert that the internal structure is indeed intact. Unfortunately, no contracts exists that will verify whether that the generated code is correct. For the time being it is mostly contracts on the requirements on how to use the compiler APIs. So if one is able to run the new code generation without contract violation, it means that one is using the compiler API correctly, but not that the generated code makes sense.

To verify that the generated code is correct, EiffelWeasel (eWeasel) is a good starting point. eWeasel is a tool that was originally written by David Hollenberg from MOSIS (a division of the University of Southern California's Information Sciences Institute)[38]. The goal of the tool is to ensure conformance of the EiffelStudio compiler with regards to the most recent language specification, as well as to detect any unforeseen regression failures from version to version.

The way eWeasel works is is that each test has a configuration file called "*tcf*", which contains a script of things to perform. For most tests, it consists of copying an Eiffel configuration file, and a set of Eiffel classes. Once copied, the compiler is instructed to compile the code. It is then possible to choose between various type of compilation, and once compiled eWeasel is instructed to run the defined scripts on the generated code (including execution of the generated executable in order to compare the output with a reference output.). If all the steps defined in the eWeasel script are working the test passes, otherwise it fails. In addition, some tests verify that the compiler reports an error, so eWeasel can also compare if the output of the compiler matches with the expected error, if not it will report this as an error.

According to Emmanuel Stapf, one would need to go up to having a 88% success rate in the tests in order to have a somewhat robust JVM implementation. Moreover, and if possible, one might need to compare the Java results against the normal results generated by the C back-end, as these should not yield any differences.

As the project did not reach to a point where regression testing was possible, the details on how the eWeasel test tool works have not been investigated further, thus this is left for future work on this project.

[38]See `http://dev.eiffel.com/Eweasel`

Chapter 6

Conclusion

6.1 Discussion

This section contains discussions on both the process of the project and the current implementing, thus reflecting on some of the key issues that was revealed during the course of doing this project.

The process of the project

The process of this project has been somewhat turbulent, starting out with the assumptions that some of the work presented in Baumgartner [1] could be partially recovered from the thesis behind it, and that a library for generating JVM bytecode already existed.

The fact that all the work done presented in Baumgartner [1] was lost, whereas nothing could be recovered, meant that everything had to be rewritten. Yet, this was not the key issue as the real problem surfaced in the terms of not having a proper description on the internals of the compiler, as discussed in a bit. Moreover, it was discovered that that work presented in Gisel [13] was missing as well. Not having a library which could be used to generate JVM bytecode was a major setback, as this meant that in order to progress with the original intend of this project, writing of a library for generating JVM bytecode was now necessary.

In line with the missing description of the compilers internal structure, a correspondence with Benno Baumgartner revealed that he also has had troubles in trying to understand the internal compiler structure. As he states, he tried to plug-in his code generator into the compiler and tried to reuse as much code as possible. The result, however, was a disaster as the missing overview of how the different parts of the compiler were interconnected forced him onto the path of trial and error. This ended up costing him too much time, whereas he was not able to complete his JVM back-end implementation. Unfortunately, this meant that this project was also lacking this part of the knowledge, so additional efforts was needed in order to collect and document this knowledge.

To gain and accumulate the missing knowledge, this meant consulting the Eiffel Software team, hence asking if they were willing to invest time in explaining how the compiler worked internally, and/or providing documentation on such. Therefore the focus of the project had now taken a second turn, meaning it was equally important to document the knowledge on how to work with the internal structure of the compiler, as to how the back-end should be implemented, since knowing how the compiler may be used is something that is preceding the work of implementing the back-end. Moreover, having a description of how the internal data structures of the compiler relates and may be queried is also something that is useful for future work on that branch of the EiffelStudio compiler.

In gaining understanding of the internal structure of the EiffelStudio compiler, a broad spectra of tools have been used, but most profound was the reverse-engineering of the source code using the built-in diagram tool found in EiffelStudio (by which it was possible to get a general overview of how the different classes were related), as well as the several correspondences and video conferences held together with Emmanuel Stapf from the Eiffel Software team.

In retrospective, it might be arguable that the work on the EiffelStudio compiler should have been halted earlier in the process, thus turning towards a "smaller" compiler, like the Gobo Eiffel Compiler[39]. This would have helped in terms of validating whether it was indeed possible to implement a back-end with a simpler interface for clients to use when compiling Eiffel to the JVM.

Implementation issues

One of the discussions held with Emmanuel Stapf was related to the reuse of the existing .NET code generator classes. To be more precise, could the existing .NET vistior and code generator classes be used in terms of generating JVM bytecode? The brainstorm resulting in a decision of trying to see how far one could go, following the path of reuse. This decision ended up costing a lot of time. There were simply too many .NET specific details sprinkled across the compiler code, which caused a chain-reaction of errors to occur. As a result, this also contributed to the fact that the implementation of the back-end was not completed.

In terms of translating Eiffel to the JVM object model, the model proposed in this thesis relies on some concepts which are not available to older platforms, like generics and such. If future work requires that older versions of the JVM must be supported, then another translation scheme will be needed in some cases. However, this has not been addressed in this thesis, as it was never the intension to support all versions of the JVM, but only verify whether the generated bytecode could be simplified as a result of the evolution of the JVM.

A description of how the renaming issue may be solved, in terms of a concrete implementation, has yet to be described. Currently, the available resources (such as articles and alike) for using the new dynamic invocation on the JVM is rather limited, and most of them are even outdated. In order to implement and support the renaming facility using

[39]See `http://www.gobosoft.com/eiffel/gobo/gec/index.html`

the `invokedynamic` bytecode instruction, some work will be needed.

In terms of improving the pace of the developing, one should use the Light EiffelBase library instead of the orginal EiffelBase library, as compiling EiffelStudio using the original libraries is slow. This is due to the fact that everything has to be compiled everytime, since no pre-compiled libraries can be used, including the entire EiffelBase. This Light EiffelBase is included on the supporting material accompanying this thesis.

6.2 Future work

The next steps in continuing the work of extending the EiffelStudio compiler, thus allowing it to compiler Eiffel source code to JVM bytecode, one would have to:

Finalize the EJBCL library: The functionality of the current version of the EJBCL is limited to supporting just the needs of this project. As such, the library should be extended to be capable of performing equally as to what is found in other libraries of same nature, such as the ASM library. In addition, an adequate test-suite should be constructed, thus allowing one to verify the correctness of the generated JVM bytecode.

Translate the remaining parts of Eiffel to Java: There are a few missing language constructs, which have not been translated in the analysis phase, such as `frozen` and `select`. In order to have full support for the Eiffel language on the JVM, a translation scheme for the remaining parts of the Eiffel language must be specified.

Look more into how `invokedynamic` can be used: As described in the prior, the documentation available on the use of `invokedynamic` is rather limited. As a consequence, more time will have to be invested in order to gain sufficient knowledge on how the actual implementation can be done.

Implement the proposed back-end using the described algorithms: Once the above tasks are completed, the next step is to implemented the JVM back-end, using the model and algorithms described in section 4.2.

Invest time in looking more into eWeasel: In order to verify and validate the generated JVM bytecode classes, the eWeasel project must be used. As such, more time will also be needed in order to utilize this framework.

Moreover, before the new JVM back-end can be put into a released version of EiffelStudio, one must further interface the debugger.

6.3 Final words

From these last sections (discussion and future work) it can be concluded that there has been a lot of obstacles during this project, which is also why some of the predefined goals at this stage has not been completed.

However, the work within this project has given a lot of knowledge on both compiler design- and construction, as well as programming language implementation and translation. In addition, this project has also presented the opportunity to work globally with a large scale research/engineering project, which has been an interesting experience.

Lastly, this project has shown that when working within research fields that has only been touched upon a bit beforehand, unexpected problems can and will occur. As such, the project has given valuable knowledge within problem solving.

Appendices

Appendix A

A comparison between the terminologies used in Eiffel and Java

Language		
Eiffel	Java	Description - in Eiffel terminology (where available)
Class	Class	An implementation of an abstract date type (ADT).
Object	Object	An instance of a certain class. Exists dynamically during run-time.
N/A	Interface	An interface has no implementation; it only has the signature or in other words, just the definition of the methods without the body. Nothing similar in Eiffel exist.
Deferred	Abstract	Both features and classes can be deferred. Features need to have no body, and classes cannot be instantiated. If a class contains a deferred feature, then the class itself must be deferred as well.
Feature	Member	Attributes and routines of a class.
Routine	Method	A feature with a body. Typically does some sort of computation on the behalf of the caller.
Attribute	Field	A feature without a body. Typically holds a single return value.
Query	Field or object method	A feature with a return value.
Command	void method	A feature without a return value.
Creation feature	Constructor	A feature used to instantiate an object.

Table A.1: A comparison between the terminologies used in Eiffel and Java (An extension the table found in section 1.5 of Baumgartner [1]).

Appendix B

The Jasmin assembly file structure

In the following, an overview of the Jasmin assembly file structure is described.

The Structure of the Jasmin Assembly File Content

The overall structure of a Jasmin file is:

Listing B.1: The overall structure of an Jasmin assembly file.

```
.source <source-file>
.class <access-spec> <class-name>
.super <class-name>
.implements <class-name>
.field <access-spec> <field-name> <descriptor> [ = <value> ]
<constructors>
<methods>
```

The structure of a constructor is:

Listing B.2: The overall structure of an Jasmin assembly file.

```
.method public <init>()V
  .limit stack 1
  .limit locals 1
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method
```

The structure of a method is:

Listing B.3: The overall structure of an Jasmin assembly file.

```
.method <access-spec> <method-spec>
  .throws <class-name>
  .limit stack stack_limit
  .limit locals locals_limit
  <statements>
  return
.end method
```

Computing the limits of the stack and locals are not required, as Jasmin will try to compute these if not defined. However, these can either be set explicitly if the programmer knows better, or found manually through the use of control-flow analysis Zhao [32].

Appendix C

The EJBCL Library

Introduction

The Eiffel Java Bytecode Library (EJBCL) is a small library which can be used for generating `.class` files. This guide describes how to used in library, and how to run Jasmin in order to generate `.class` files from the output model. Note that this document does not explain the JVM itself, nor does it give a detailed explaination for the instructions known by the JVM.

Working with the library

Creating a New Java class

When working with generating bytecode and `.class` files, creating either an interface or a concrete class is one of the most fundamental tasks.

Listings C.1 gives an example on how to generate a class named `Foo`, which contains no fields or methods, using the EJBCL:

Listing C.1: Generating the class `Foo` using EJBCL.

```
class EJBCL_TEST
inherit
  JAVA_OPCODE
  JAVA_CONSTANTS

feature
  generate_class
      -- Generate a new public class named 'Foo'
```

```
        local
          l_class: JAVA_CLASS
          l_class_name: STRING
        do
            --| Set the name of the class.
          l_class_name := "Foo"

            --| Generate the class.
          create l_class.make (V_1_7,
                    acc_public,
                    l_class_name,
                    "java/lang/Object", Void)
        end
    end
```

Adding a New Field

In order to add a field to the class `Foo`, one must create a new field using the `FIELD` class as shown in listings C.2:

Listing C.2: Adding a new field to class `Foo`.

```
    feature
      generate_class
        local
          l_class: JAVA_CLASS
          l_class_name: STRING
          l_field: FIELD
          l_field_name: STRING
          l_type_descriptor: STRING
        do
            --| Code omitted...
            --| Create a new public field of type ’String’ named ’Bar’
          l_field_name := "Bar"
          l_type_descriptor := "Ljava/lang/String"
          create l_field.make (l_class,
                    acc_public,
                    l_field_name,
                    l_type_descriptor, Void)

            --| Add the field to class ’Foo’
          l_class.fields.put (l_field)
        end
```

Adding a New Method

In order to add a new method to the class `Foo`, one must create a new method using the `METHOD` class as shown in listings C.3:

Listing C.3: Adding a new method to class Foo.

```
feature
  generate_class
    local
      l_class: JAVA_CLASS
      l_class_name: STRING
      l_field: FIELD
      l_field_name: STRING
      l_method: METHOD
      l_type_descriptor: STRING
    do
        --| Code omitted...
        --| Create a new method with the signature
        --| 'void m(float f, int i)'
      l_type_descriptor := "(FI)V"
      create l_method.(l_class,
              acc_public,
              "m",
              l_type_descriptor)

        --| Instructions to the method's body is added here...
      instructions.extend (create {INSTRUCTION}.make ({JAVA_OPCODES}.
          return))

        --| Add the method to class 'Foo'
      l_class.methods.put (l_field)
    end
```

Generating JVM Bytecode

Once the output model has been defined, the last step is to produce and emit Jasmin assembly code. As shown in listings C.4 on page 82, this is done by using the `JASMIN_ASSEMBLY_GENERATOR` visitor class.

Listing C.4: Generating Jasmin assembly code for class Foo.

```
feature
  generate_assembly_code (a_class: JAVA_CLASS)
    local
      l_assembly_generator: CLASS_ITEM_VISITOR
      l_output: STRING
    do
        --| In order to emit Jasmin assembly code,
        --| the visitor class is instatiated as a
        --| Jasmin code generator
      create {JASMIN_ASSEMBLY_GENERATOR} l_assembly_generator.make

        --| Instruct the class to be processed using the
        --| Jasmin assembly generator
      a_class.accept (assembly_generator)

        --| Get the generated Jasmin assembly code
      l_output := assembly_generator.out
  end
```

Once the output has been generated, the Jasmin assembler may be used to generate JVM bytecode, using the command:

```
java -jar jasmin.jar Foo.j
```

This will generate a new `.class` file named `Foo.class`, which can be used inside the JVM or from inside another Java program.

Code Generation Workflow

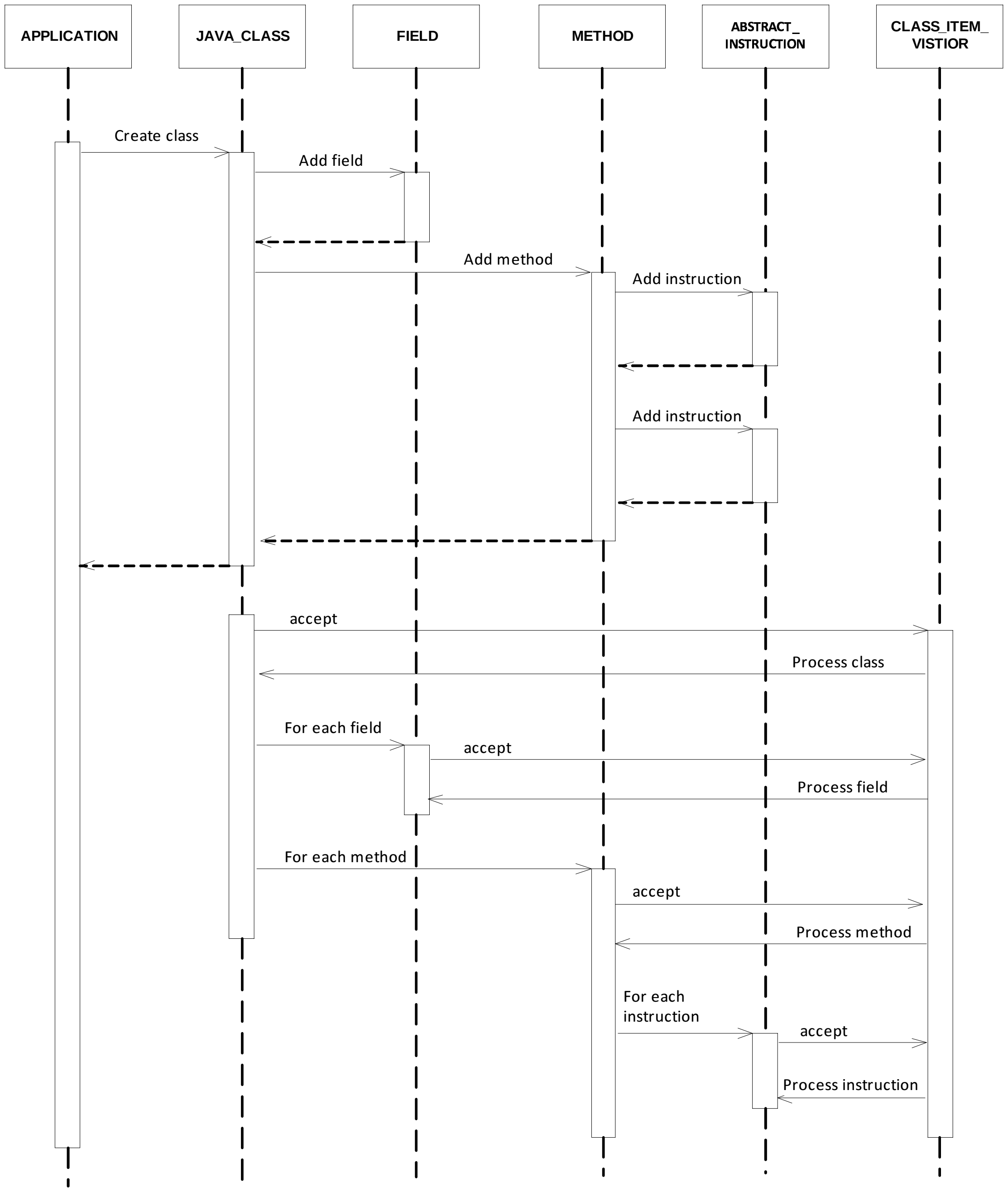

Figure C.1: Workflow diagram code generation using class using the UML notation

References

[1] Benno Baumgartner. Jeiffel - a java bytecode generator backend for the ise eiffel compiler. diploma thesis, ETH, Eidgenössische Technische Hochschule Zürich, Chair of Software Engineering, 2005.

[2] ECMA International. Eiffel: Analysis, design and programming language. `http://tinyurl.com/cq8gw`, 2006.

[3] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. ISBN 0136290493.

[4] Raphael Simon, Stapf Emmanuel, and Bertrand Meyer. Full eiffel on the .net framework. `http://tinyurl.com/6bhlkmp`, 2002.

[5] Dominique Colnet and Olivier Zendra. Targeting the java virtual machine with genericity, multiple inheritance, assertions and expanded types.

[6] Tim Lindholm, Frank Yellin, Bracha Gilad, and Buckley Alex. *Java Virtual Machine Specification*. Oracle America, Inc. and/or its affiliates, java se 7 edition edition, 2012.

[7] B.J. Evans and M. Verburg. *The Well-Grounded Java Developer: Java 7 and Polyglot Programming on the JVM*. Manning Pubs Co Series. Manning Publications, 2012. ISBN 9781617290060.

[8] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.

[9] Bertrand Meyer. Eiffel: The essentials.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006. ISBN 0321486811.

[11] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009.

[12] Eric Bruneton. Asm 4.0, a java bytecode engineering library.

[13] Daniel Gisel. Eiffel library to generate java bytecodes. diploma thesis, ETH, Eidgenössische Technische Hochschule Zürich, Professur für Software Engineering /Chair of Software Engineering, 2003.

[14] Marco Trudel, Manuel Oriol, Carlo A. Furia, and Martin Nordio. Automated translation of java source code to eiffel. In *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 20–35. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21951-1.

[15] Keith Ward. No java required: Write android apps in c#. `http://tinyurl.com/cb7ttst`, 2012.

[16] YangSun Lee and Seungwon Na. Java bytecode-to-.net msil translator for construction of platform independent information systems. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3215 of *Lecture Notes in Computer Science*, pages 826–832. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-23205-6.

[17] Joshua Engel. *Programming for the Java Virtual Machine with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201309726.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612.

[19] Joshua Fox. When is a singleton not a singleton? `http://tinyurl.com/7tvrd`, 2001.

[20] Peter Haggar. Double-checked locking and the singleton pattern - a comprehensive look at this broken programming idiom. `http://tinyurl.com/66odeut`, 2002.

[21] Bertrand Meyer, Cover Design, and Rich Ayling. An eiffel tutorial, 2001.

[22] Erik Poll, Patrice Chalin, David Cok, Joe Kiniry, and Gary T. Leavens. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS*, pages 342–363. Springer, 2006.

[23] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications, 2003.

[24] David Pollak. Faking closures on the jvm isn't as simple as it looks. `http://tinyurl.com/d9tvyr3`, 2010.

[25] Oracle. Jsr 335: Lambda expressions for the javatm programming language. `http://tinyurl.com/bszvmnb`, 2012.

[26] K. Waldén and J.M. Nerson. *Seamless object-oriented software architecture: analysis and design of reliable systems*. Prentice Hall object-oriented series. Prentice Hall, 1995. ISBN 9780130313034.

[27] Kim Waldén. Business object notation (bon). `http://tinyurl.com/cr85rpp`, 1998.

[28] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[29] Eiffel Software. Eiffelstudio integrated development environment. `http://tinyurl.com/67k5tw`.

[30] Peter Sestoft. Programming language concepts for software developers, 2010.

[31] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle America, Inc. and/or its affiliates, java se 7 edition edition, 2012.

[32] Jianjun Zhao. Analyzing control flow in java bytecode. In *Proc. 16th Conference of Japan Society for Software Science and Technology*, pages 313–316, 1999.