# Extending the Java Programming Language with Generators

*Master's thesis*

Jonathan Guzman Carmona

# Extending the Java Programming Language with Generators

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jonathan Guzman Carmona
born in Cali, Colombia

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl


TOPdesk B.V.
Martinus Nijhofflaan 2, 13th floor
2624 ES
Delft, the Netherlands
www.topdesk.nl

# Extending the Java Programming Language with Generators

Author: Jonathan Guzman Carmona
Student id: 1335596
Email: j.guzmancarmona@student.tudelft.nl

**Abstract**

The Java programming language allows to create portable applications in a variety of domains. With the continuous development and demanding environment in industrial and research fields many proposals exist to extend the language in order to facilitate a more easier development and implementation of applications. Many extensions have been implemented by applying several program transformation techniques such as Domain Specific Languages (DSLs), extensions to existing compilers, language extension assimilation, intermediate code transformation and strategy rewriting frameworks. A particular extension that has not yet been integrated in the Java programming language and merits further research is generators. This extension allows an easier implementation of iterators and is suitable for many other patterns due to its semantics. In this thesis report we introduce generators and discuss the design and implementation of a non-intrusive solution that extends the Java programming language with this construct by means of intermediate code manipulation (bytecode weaving). We also evaluate the implemented solution and demonstrate a sample application where we assess the performance of generators. Finally, we discuss our experiences of implementing this extension in relation to a solution for language extensions in general by means of this non-intrusive approach.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | MSc. L.C.L. Kats, Faculty EEMCS, TU Delft |
| University co-supervisor: | Dr. E. Visser, Faculty EEMCS, TU Delft |
| Company supervisor: | BSc. R. Spilker, TOPdesk B.V. |
| Committee Member: | Prof. Dr. K.G. Langendoen, Faculty EEMCS, TU Delft |

# Preface

Ever since I started studying computer science, I have always had a special interest for programming languages. I remember that I followed the courses of concepts of programming languages and compiler construction with lots of enthusiasm and interest during my bachelor of computer science at the Leiden University. I am glad to have continued my study by following the master computer science at the Delft University of Technology. It is here that I got the chance to gain a more in depth understanding in the exciting field of software engineering. This master thesis project has given me the opportunity to work on a subject of personal interest where I have been able to put all my knowledge of programming languages into practice.

I would like to thank the people who have contributed their efforts in bringing this work together. Lennart C.L. Kats for his guidance, useful comments, remarks and critical view on this work. Roel Spilker who provided guidance, useful feedback and support at TOPdesk. Eelco Visser for his support and remarks. Arie van Deursen for charing the graduation committee and Koen Langendoen for participating in it. Finally, I would like to thank my brothers Alex and Steven for supporting me. Glenn Martheze for giving me advise and support whenever I needed. Specially, I would like to thank my mother without whom I would have not had the chance to follow a study.

Jonathan Guzman Carmona
Delft, the Netherlands
May 18, 2009

# Contents

# List of Figures

# Chapter 1

# Introduction

The Java programming language [27] since its first introduction by Sun Microsystems [12] in 1995, has become one of the most widely used languages for general purpose programming. It has been used for the implementation of systems in a wide range of domains. Furthermore Java has, since then, been improved and extended. The language was designed to be extended in the future [28]. In 1998, the process of extending the language was formalized and called the Java Community Process (JCP). Allowing interested parties to be involved in the definition of future versions and features of the Java platform.

Besides Sun, third parties along with academic research have tried to extend Java with a certain purpose in mind [37, 33, 34, 30]. In particular many techniques from this research have been developed for the extension of the Java platform with new features. Many of these techniques include program transformation. Program transformation has had a lot of attention from researches the last years. Compiler extension techniques, language extension assimilation, intermediate code transformation and Domain-Specific languages are some of the techniques used.

The rise of dynamic languages for rapid application development like Python [8] and Ruby [9] have its share pushing productivity and ease of use to use to a higher level. These programming languages come with features and programming constructs that prove to be very productive. Lots of attempts have been made to port certain extensions to the Java platform and there are already implementations for Python and Ruby for use in the Java platform [6, 5]. Many of these languages have a useful construct called generators. This construct allows to create iterators which are easier to implement and use. Some attempts (see Section 4.2) have been made to port this particular construct to the Java platform, but without a wide use and acceptance within the Java community.

The main aim of this document is to provide an overview of a solution to port the generator construct into the Java programming language which constitutes a thesis project for the master Computer Science at the Delft University of technology.

## 1.1 TOPdesk

The thesis project carried out is an assignment from TOPdesk. TOPdesk [18] is a company that strives for the standardization of rich information and intensive knowledge based processes in organizations, which is supported by user friendly software where humans are the central key in the process. To this aim, TOPdesk has developed a service management application called TOPdesk. This tool provides support for automation, facility management, complaints registration, service desk or service support within organizations. The software and services enables organizations to efficiently organize their service delivery. The solutions provide by TOPdesk are focused on service desks that support employees, business relationships, consumers and citizens.

The development department of TOPdesk consists of several small teams that focus on different modules of the TOPdesk application. TOPdesk chose to implement the application in the Java programming language as a web based application which provides a lot of flexibility to their customers and to TOPdesk itself for the further development of the application. One development team in particular is responsible for the technical part of the application server. This includes the database, webserver and application service framework. This team is also responsible for doing research on new technologies that facilitates the further development of new applications and improvement of already existing features. It is in collaboration with this team that the thesis project has been carried out.

## 1.2 Problem Statement

Generators facilitate an easier implementation of iterators and their usage. The implementation is easier since the traversal algorithm can be written without worrying to much about keeping any state variables (see Section 2.3). TOPdesk's development team is always looking for enhancements that facilitates cleaner code that results in less bugs, higher performance, leading to a more productive environment. Since the Java programming language does not have the generator construct, the development team would like to port this construct into the Java programming language.

TOPdesk is a commercial company using standard tools for development like Sun's Java compiler, the Eclipse IDE and many other standards tools, posing several constraints that need to be met in order to port the generator construct into the Java programming language successfully. These constraints can be formulated as the following research question:

**Q1:** How to non-intrusively extend the Java programming language with a generator construct?

Non-intrusive means that the introduction of this new construct should not pose any restrictions to the tooling being used in the standard language and the environment in which it is used. As seen in literature [30, 34, 33, 37], Java has been exposed

to experiments aiming at extending the language with some new feature. Many of the extensions attempts seen in literature are experimental and cannot be used in a production environment. The challenge here is to port a construct in a production environment. It is essential to still be able to use the standard Java compiler provided by Sun and not break any existing code. Furthermore, it should not impede developers from using any of the conventional tools for development such as IDEs, debuggers and profilers.

The previous question can be put into a broader context focusing on new language extensions in general. This leads to the following research question:

**Q2:** How to develop a language extension so it can be successfully introduced in a non-intrusive manner?

This thesis project focuses on the implementation of an extension with a construct such that it can be used in a production environment. This means that there is a need to identify a mechanism to extend the Java programming language with a construct in a non-intrusive manner as described earlier for the first research question. By carrying out this project, we strive to gain theoretical and practical knowledge that can hopefully give us an insight into developing a methodology that can answer this research question.

## 1.3 Outline

The remainder of this thesis report is as follows. Chapter 2 provides a detailed overview of the Java platform including the Java virtual machine and its instruction set. Furthermore the Iterator design pattern and generators are explained in detailed along with examples of its usage. Chapter 3 gives an overview of program transformation techniques and systems. Chapter 4 discusses the design space of this thesis in terms of the requirements, existing solutions and a proposed solution along with a motivation for the design choices. Chapter 5 discusses the architecture of the implemented solution. Chapter 6 provides a discussion on how the implemented solution was evaluated. Chapter 7 reflects on the findings and lessons learned during this thesis work and puts the problem statement into a more broader context by giving an outline of a possible solution for integrating arbitrary extensions into the Java programming language in a non-intrusive manner. Finally, chapter 8 concludes this document and discusses possible future work.

# Chapter 2

# Background and Preliminaries

This chapter will introduce the Java platform along with its encompassing technologies. The Java Virtual Machine (JVM) and its bytecode instruction set will be explained along with the Java class file format as these form the core of the Java platform. Furthermore, a discussion on the Java programming language will be provided. Finally, the Iterator design pattern will be explained as it is implicitly supported by generators. Generators will be explained as well along with some examples to show its usefulness.

## 2.1 The Java Platform

The Java platform is a technology developed by Sun Micro Systems [12]. It consists of a Java Virtual Machine (JVM), a core Application Programming Interface (API), and the compiler that creates the bytecode for some programming language targeting to run on the JVM. Sun developed the Java programming language as the host language for the JVM, but nowadays other programming languages have been developed or adapted to run on the JVM as well such as Scala [36], Jython [6] and JRuby [5]. The Java Virtual Machine or JVM is often referred as the Java platform which is an environment where all program run on. Furthermore the Java platform provides a set of tools to develop applications like debuggers and a launcher to start the applications to be run on an instance of the JVM.

### 2.1.1 The Java Virtual Machine

According to Sun's definition [4], a platform is the hardware or software environment in which a program runs. Many existing platforms (Windows, Linux, Mac OS) can be described as a combination of the operating system and the underlying hardware. The Java Virtual Machine [31] is the base for the Java platform and is ported onto various hardware-based platforms. The concept of the Java platform is to have a platform-independent environment with security in mind. The Java platform differs from other platforms in that it is just software that runs on top of the other hardware-based platforms. This concept was realized by the virtual machine. This means that each host operating system needs its own implementation of the JVM and runtime (see Figure 2.1).

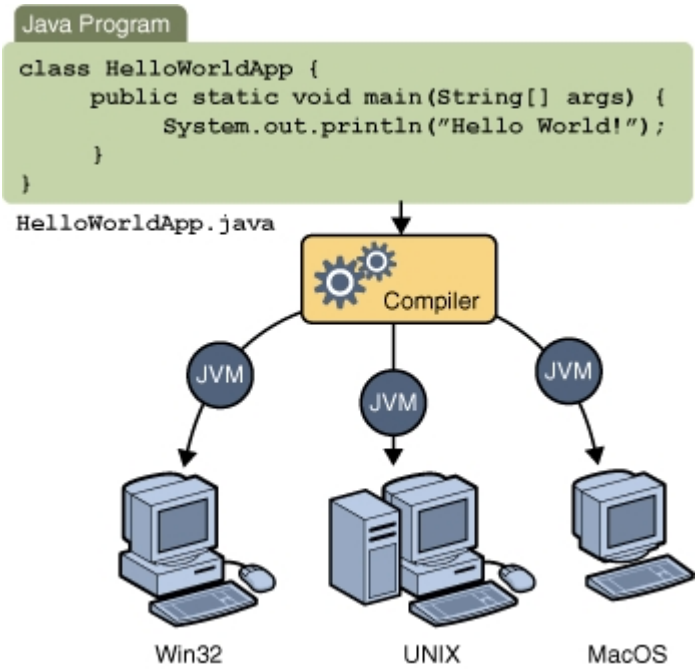


**Figure 2.1:** Relation between the JVM and several operating systems. Taken from [42]

The Java virtual machine is an abstract machine. It is the component of the Java platform technology responsible for its hardware- and operating system-independence. It has an instruction set (the bytecode instruction set) and manipulates various memory areas at run time (see Figure 2.2). It supports a particular binary format, the class file format (more on this later). A class file contains Java virtual machine instructions (bytecodes) and a symbol table, as well as other ancillary information. The Java virtual machine operates on bytecode and has a stack-based architecture [24, 10]. It imposes strong format and structural constrains on the code in a class file. It allows very fine-grained control over the actions that code within the machine is permitted to take. It does this to provide security and protect user from malicious programs.

The JVM runtime executes .class or .jar files by emulating the JVM instruction set or by using a just-in-time-compiler (JIT) such as Sun's HotSpot [11]. JIT compiles bytecode at runtime prior to executing it natively. This helps improve performance over interpreters. The improvement comes from caching results of translating blocks of bytecode and not simply re-evaluating each line or operand each time it is met. Additionally, the JVM verifies all bytecode before it is executed to protect certain functions and data structures belonging to "trusted" code from access or corruption by "untrusted" code executing within the same JVM.

The advantages of the JVM approach is that any language with functionality that can be

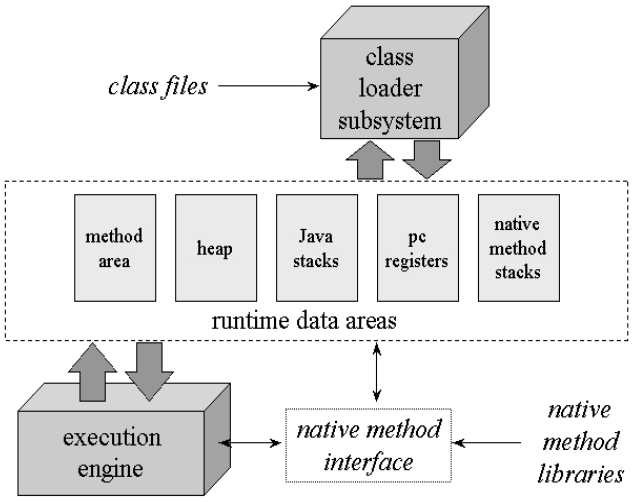


**Figure 2.2:** JVM internal architecture. Taken from [42]

expressed in terms of a valid class file can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementors [6, 5] of other languages are turning to the Java virtual machine as a delivery vehicle for their languages. Originally, the JVM was primarily aimed at running compiled Java programs (see next section), but recently, as dynamic languages [8, 9] have grown in interest, built support for dynamic languages is under research and development [2].

### 2.1.2 The Java Programming Language

The Java programming language is a general-purpose concurrent class-based object-oriented programming language, specifically designed to have as few implementation dependencies as possible. It allows application developers to write a program once and then be able to run it everywhere on the Internet [27].

Java is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, the design has avoided including new and untested features. The Java programming language is strongly typed which allows for early detection of as many errors as possible during compile-time. The Java programming language is a relatively high-level language, in that details of the machine representation are not available through the language. It includes

automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's free or C++'s delete).

The Java programming language is normally compiled to the bytecode instruction set and binary format as defined in the JVM specification [31]. The Java virtual machine, discussed in the previous section, was designed to support the Java programming language.

The language has been used to develop applications in a wide range of domains and platforms such as desktops, servers and consumer devices. The Java platform along with the Java programming language has grown to include the portfolio of the following platforms:

- **The Java Platform Standard Edition (Java SE):** provides an environment for Core Java and Desktop Java applications development. It is the basis for the Java Platform Enterprise Edition and Java Web Services technologies. It has the compiler, tools, runtimes, and Java APIs that let you write, test, deploy and run applications.
- **The Java Platform Enterprise Edition (Java EE):** defines the standard for developing component-based multitier enterprise applications. It is based on Java SE and provides additional services, tools and APIs to support simplified enterprise applications development.
- **The Java Platform Micro Edition (Java ME):** is a set of technologies and specifications targeted at consumer and embedded devices, such as mobile phones, personal digital assistants (PDA's), printers, and TV set-top boxes.
- **Java Card Technology:** Java platform adaptation to enable smart cards and other intelligent devices with limited memory and processing capabilities to benefit from many advantages of Java technology.

### 2.1.3 The Java Class File Format

Code to be executed by the Java virtual machine must be compiled into a representation that uses a hardware- and operating system-independent binary format. This compiled code is typically (but not necessarily) stored in a file and is known as the class file format [31]. The class file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first.

The structure of a compiled class (containing a class or interface definition) is simple. Unlike natively compiled applications, it retains the structural information and almost all the symbols from the source code. A compiled class contains the following which is depicted in Figure 2.3:

| | |
|---|---|
| Modifiers, name, super class, interfaces | |
| Constant pool: numeric, string and type constants | |
| Source file name (optional) | |
| Enclosing class reference | |
| Annotation* | |
| Inner class* | |
| Field* | Modifiers, name, type |
| | Annotation* |
| | Attribute* |
| Method* | Modifiers, name, return and parameter types |
| | Annotation* |
| | Attribute* |
| | Compiled code |

**Figure 2.3:** Overview of the structure of the class file format. Taken from [22].

- A modifiers description section (such as `public` or `private`), the name, the super class, the interfaces and the annotations of the class.
- One section per declared field in the class. Where each section describes its modifiers, the name, type and annotations of a field.
- One section per method and constructor declared in the class. Where each section describes the modifiers, the name, the return, and parameter types, and the annotations of a method. Additionally it contains the compiled code of the method as a sequence of Java bytecode instructions.

There are some differences between source and compiled classes. A compiled class describes only one class, while a source file can contain several classes. The other classes are compiled into a different class file each containing a reference to the "main" class or enclosing method[1]. This "main" class contains in turn a reference to its inner classes. Furthermore, a compiled class does not contain comments, but can contain class, field, method and code attributes[2]. A compiled class does not contain a package and import section, so all type names must be fully qualified.

Additionally, a compiled class contains a constant pool section. This pool is an array containing all the numeric, string and type constants that appear in the class. These constants are defined only once, in the constant pool section, and are referenced by their index in all other sections of the class file. An important difference between a source an compiled classes is that Java types are represented differently in compiled and source classes. These

[1] In case of an inner class defined inside a method.

[2] These attributes can be used to associate additional information to these elements. However, since the introduction of annotations in Java 5, attributes have become mostly useless.

| 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; |

**Figure 2.4:** Type descriptors of some Java types. Taken from [22].

| Method declaration in source file | Method descriptor |
|---|---|
| void m(int i, float f) | (IF)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; |

**Figure 2.5:** Sample method descriptors. Taken from [22]

types are represented with internal names. For instance, the internal name of a class is just its qualified name of this class, where dots are replaced with slashes (i.e. the internal name of `String` is `java/lang/String`).

Internal names are used only for types constrained to class or interface types. Other types are represented with type descriptors. For primitive types the descriptors are single characters (see also Figure 2.4). 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.

As already mentioned, a compiled class contains a method descriptor. This is a list of type descriptions that describe the parameter types and the return type of a method, in a single string (see Figure 2.5). A method descriptor starts with a left parenthesis, followed by the type descriptors of each formal parameter, followed by a right parenthesis, followed by the type of the descriptor of the return type or `V` if the method returns `void` [3].

[3] A method descriptor does not contain the method's name or the argument names.

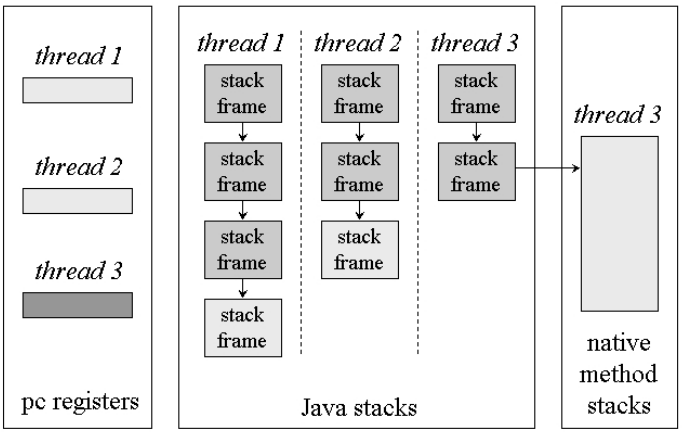


**Figure 2.6:** Runtime data areas exclusive to each thread. Taken from [42]

### 2.1.4 The JVM Instruction Set

A Java virtual machine instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation [31]. The number and size of the operands are determined by the opcode. If an operand is more than one byte in size, then it is stored in big-endian order high-order byte first. The decision to limit the JVM opcode to a byte and to forgo data alignment within compiled code reflects a conscious bias in favor of compactness, limiting the instruction set size.

Before we dive into the different bytecode instructions it is necessary to discuss the Java Virtual Machine execution model. Java code is executed in threads. Each thread has its own execution stack, which is made of frames. Each frame represents a method invocation. Each time a method is invoked, a new frame is pushed on the current thread's execution stack. When the method returns, either normally or because of an exception, this frame is popped from the execution stack and execution continues in the calling method (whose frame is now on top of the stack). Figure 2.6 illustrates this concept.

Each frame contains two main parts: a local variables part and an operand stack part. There is additionally a frame data part with a reference to the the runtime constant pool of the class of the current method. This frame data part is also used for method invocation completion (for more details see JVM 3.6.4 and 3.6.5). The local variables part contains variables that can be accessed by their index, in random order. The local variables part is namely organized as a zero-based array. Any instruction using a value from the local vari-

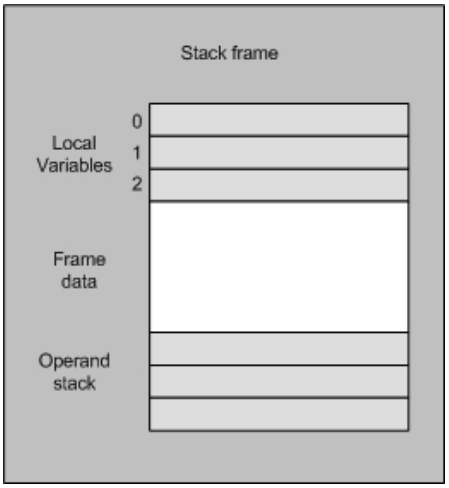


**Figure 2.7:** Anatomy of a frame.

ables section provide an index into the zero-based array. The operand stack part is organized as an array of words. But accessed by pushing and popping values. The size of the local variables and operand stack parts depends on the method's code. It is computed at compile time and is stored along with the bytecode instructions in compiled classes. Figure 2.7 shows this concept. This frame belongs to a method with an operand stack size of three. The amount of variables is two in the case of an instance method (since slot 0 is reserved for `this`. In the case of a static method the amount of variables held by this stack frame would be three.)

A bytecode instruction is made of an opcode that identifies this instruction and a fixed number of arguments. The opcode is an unsigned byte value and is identified by a mnemonic symbol (i.e. opcode value 0, designed by mnemonic symbol NOP, which is the instruction that does nothing). The arguments are static values defining precise instruction behaviour. Which are given after the opcode. For instance the `GOTO` *`label`* instruction, takes as argument *`label`*, a label that designates the next instruction to be executed [4].

Most of the instructions in the Java virtual machine instruction set encode type information about the operations they perform. The majority of the typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter (I (integer), L (long), S (short), B (byte), C (character), F (float), D (double), A (address or reference)). Some instructions for which the type is unambiguous do not have a type letter in their mnemonic (i.e. ARRAYLENGTH). It should be noted that not all instructions have the forms for the integral types byte, char, and short. None have forms for the boolean type. Compilers

[4]Instruction arguments must not be confused with instruction operands. Argument values are namely statically known and stored in the compiled code, while operand values come from the operand stack and are only known at runtime.

encode loads of literal values of types byte, short using Java virtual machine instructions that sign-extend those values to values of type `int` at compile time. Loads of literal values of types `boolean` and `char` are encoded using instructions that zero-extend the literal to a value of type int at compiler time or run time. The same goes for arrays of previous types.

Bytecode instructions can be divided in two categories. A small set of instructions is designed to transfer values from the local variables to the operand stack, and vice versa. The other instructions only act on the operand stack. For instance popping values, computing a result on these values, and pushing it back on the stack.

### Load and Store Instructions

The `ILOAD, LLOAD, FLOAD,` and `ALOAD` instructions read a local variable and push its value on the operand stack. They take as argument the index `i` of the local variable that must be read. `ILOAD` is used to load a `boolean, byte, char, short,` or `int` local variable. `LLOAD, FLOAD` and `DLOAD` are used to load a `long, float` or `double` value, respectively[5]. Finally `ALOAD` is used to load any non primitive value (object and array references). Symmetrically the `ISTORE, LSTORE, FSTORE, DSTORE,` and `ASTORE` instructions pop a value from the operand stack and store it in a local variable designated by its index `i`.

### Stack

These instructions are used to manipulate values on the stack. `POP` pops the value on top of the stack, `DUP` pushes a copy of the top stack value, `SWAP` pops two values and pushes them in the reverse order, etc.

### Constants

These instructions push a constant value on the operand stack. `ACONST_NULL` pushes `null`, `ICONST_0` pushes the int value `0`, `FCONST_0` pushes `0f`, `DCONST_0` pushes `0d`, `BIPUSH` *b* pushes the byte value *b*, `SIPUSH` *s* pushes the short value *s*, `LDC` *cst* pushes the arbitrary `int, float, long, double, String,` or `class` constant *cst*, etc.

### Arithmetic and Logic

These instructions pop numeric values from the operand stack combine them and push the result on the stack. They do not have any argument. `xADD, xSUB, xMUL, xDIV` and `xREM` correspond to the +, -, *, / and % operations, where `x` is either `I, L, F` or `D`. Similarly there are other instructions corresponding to `< <, > >, > > >, |, &` and `^`, for `int` and `long` values.

[5] `LLOAD` and `DLOAD` actually load the two slots `i` and `i` + 1.

### Casts

These instructions pop a value from the stack, convert it to another type, and push the result back. They correspond to cast expressions in Java. `I2F, F2D, L2D,` etc. convert numeric values from one numeric type to another. `CHECKCAST` t converts a reference value to the type `t`.

### Objects

These instructions are used to create objects, lock them, test their type, etc. For instance the `NEW` *`type`* instruction pushes a new object of type *`type`* on the stack (where `type` is an internal name).

### Fields

These instructions read or write the value of a field. `GETFIELD` *`owner name desc`* pops an object reference, and pushes the value of its `name` field. `PUTFIELD` *`owner name desc`* pops a value and an object reference, and stores this value in its `name` field. In both cases the object must be of type *`owner`*, and its field must be of type *`desc`*. `GETSTATIC PUTSTATIC` are similar instructions, but for static fields.

### Methods

These instructions invoke a method or a constructor. They pop as many values as there are method arguments, plus one value for the target object, and push the result of the method invocation. `INVOKEVIRTUAL` *`owner name desc`* invokes the name method defined in class *`owner`*, and whose method descriptor is *`desc`*. `INVOKESTATIC` is used for static methods, `INVOKESPECIAL` for private methods and constructors, and `INVOKEINTERFACE` for methods defined in interfaces.

### Arrays

These instructions are used to read and write values in arrays. The `xALOAD` instructions pop an index and an array, and push the value of the array element at this index. The `xASTORE` instructions pop a value, an index and an array, and store this value at that index in the array. Here `x` can be `I(int), L(long), F(float), D(double)` or `A(object or reference)`, but also `B(byte), C(char)` or `S(short)`.

### Jumps

These instructions jump to an arbitrary instruction if some condition is true, or unconditionally. They are used to compile `if, for, do, while, break` and `continue` instructions. For instance `IFEQ` label pops an `int` value from the stack, and jumps to the instruction designed by label `if` this value is `0` (otherwise execution continues normally to the next instruction). Many other jump instructions exist, such as `IFNE` or `IFGE`. Finally `TABLESWITCH` and `LOOKUPSWITCH` correspond to the switch Java instruction.

### Exceptions

The Java programming language uses exceptions to handle errors and other exceptional events. These errors and other exceptional events are violations to semantic constraints. An example of such a violation is an attempt to index outside the bounds of an array. The JVM signals this error to the program as an exception. This causes a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. The Java language programming provides a `try/catch/finally` syntax to this aim (see JLS [27]). An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred (the exception handler). Programs can also throw exceptions explicitly using throw statements.

In bytecode an exception is thrown using the `ATHROW` instruction. This instruction pops the exception object that is on top of the stack. In classes compiled for version of Java less or equal to 1.5, the implementation of the `finally` keyword uses `jsr, jsr_w,` (jump to subroutine) and `ret` (return from subroutine) instructions. `Finally` is compiled as a subroutine within the JVM code for its method, much like an exception handler. When a `jsr` instruction that invokes the subroutine is executed, it pushes its return address of the instruction after the `jsr` that is being executed, onto the operand stack as a value of the type `returnAddress`. The code for the subroutine stores the return address in a local variable. At the end of the subroutine, a `ret` instruction fetches the return address from the local variable and transfers control to the instruction at the return address. Classes compiled for Java version 1.6 do not contain these instructions. They have been removed to simplify the new verifier architecture introduced in Java 6. This was possible because they are not strictly necessary.

### Synchronization

The Java virtual machine supports synchronization of both methods and sequences of instructions within a method using a single synchronization construct, the *monitor*. Method-level synchronization is handled as part of method invocation and return. Synchronization of instructions is typically used to encode the synchronized blocks of the Java programming language. The Java virtual machine supplies the `monitorenter` and `monitorexit` instructions to support such constructs.

### Frames

In order to speed up the class verification process inside the JVM, classes compiled for Java 6 or higher contain, in addition to bytecode instructions a set of `stack map frames`. A stack map frame gives the state of the execution frame of a method at some point during its execution. More precisely it gives the `type` of the values that are contained in each local variable slot and in each operand stack slot just before some bytecode instruction is executed. Furthermore, In order to save space, a compiled method does not contain one frame per instruction. In fact it contains only the frames for the instructions that correspond

to jump targets or exception handlers, or that follow unconditional jump instructions as the other frames can be easily and quickly inferred from these ones.

#### Return

Finally the `xRETURN` and `RETURN` instructions are used to terminate the execution of a method and to return its result to the caller. `RETURN` is used for methods that return `void`, and `xRETURN` for the other methods.

#### Support for Debugging

Sun's Java compiler allows to compile Java programs generating additional debug information in different levels. It uses the following options:

**-g** Generates all debugging information, including local variables.

**-g:none** Do not generate any debug information.

**-g:**{*Keywordlist*} Generate the specified information. Which is a list of comma separated list of key words. Valid keywords are:

**source** Source file debugging information

**lines** Line number debugging information

**vars** Local variable debugging information

The Java virtual machine instruction set has a very simple mechanism to support debug information. It introduces two tables for debug information in all member methods: the line number table and the the local variable table. Furthermore, it inserts a label that designates a group of instructions as belonging to a line number in the source code. So the line number table consists of a mapping of line numbers in the source code to labels in the bytecode (see Figure 2.8). The local variable table contains information about the start and end labels in which the variable is visible (scope), the slot in the local variables part of the frame, the name in the source code, and finally its signature (its type) (see Figure 2.9).

```
1 LineNumberTable:
2 line 15: L3
3 ...
4 line 18: L1
5 ...
6 line 20: L2
```

**Figure 2.8:** Example of a Line number table in bytecode.

To illustrate the usage and applicability of the bytecode instructions, Figure 2.10 shows the typical "Hello World" example in Java. Figure 2.11 shows the compiled code of the

```
1  LocalVariableTable:
2  Start  Length  Slot  Name  Signature
3   L3     L10     0    this  <qualified class name>
4   L3     L10     1     a    I
5   L4     L10     2     b    I
```

**Figure 2.9:** Example of a Local variable table in bytecode.

```
1  public class HelloWorld {
2    static final String GREETING = "HELLO WORLD";
3
4    public static void main(String[] args) {
5      System.out.println(GREETING);
6    }
7  }
```

**Figure 2.10:** Hello world example in Java.

"Hello world" example from the generated class file[6]. As we can see there is a constant pool section with all string constants, the declaration of the static field `GREETING`, the implicit constructor generated by the compiler and the main method.

## 2.2 The Iterator Design Pattern

The iterator design pattern's intent is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It is also known as a cursor and is the category of object behavioural patterns [26]. The key idea in this pattern is to take the responsibility for access and traversal out of the aggregate object and put it into an iterator object. The iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element. Meaning that it knows which elements have been traversed already.

The iterator design pattern can be used to access an aggregate object's content without exposing its internal representation and it supports multiple traversals of aggregate objects. Furthermore, it provides an uniform interface for traversing different aggregate structures (polymorphic iteration).

The iterator pattern has three important consequences:

1. **It supports variations in the traversal of an aggregate:** complex aggregates may be traversed in many ways. Iterators make it easy to change the traversal algorithm.

[6]This is obtained by running the javap tool with the following options: `-c -s -l -verbose -private -classpath <classpath> <qualified class name>`.

```
1  Compiled from "HelloWorld.java"
2  public class HelloWorld extends java.lang.Object
3    SourceFile: "HelloWorld.java"
4    minor version: 0
5    major version: 50
6    Constant pool:
7  const #1 = Method       #6.#18; // java/lang/Object."<init>":()V
8  const #2 = Field        #19.#20;// java/lang/System.out:Ljava/io/PrintStream;
9  ...
10 const #30 = Asciz       println;
11 const #31 = Asciz       (Ljava/lang/String;)V;
12
13 {
14 static final java.lang.String GREETING;
15   Signature: Ljava/lang/String;
16   Constant value: String HELLO WORLD
17
18 public HelloWorld();
19   Signature: ()V
20   Code:
21     Stack=1, Locals=1, Args_size=1
22     0:   aload_0
23     1:   invokespecial   #1; // Method java/lang/Object."<init>":()V
24     4:   return
25   LineNumberTable:
26     line 2: 0
27
28 public static void main(java.lang.String[]);
29   Signature: ([Ljava/lang/String;)V
30   Code:
31     Stack=2, Locals=1, Args_size=1
32     0:   getstatic       #2; // Field java/lang/System.out:Ljava/io/PrintStream;
33     3:   ldc     #3; //String HELLO WORLD
34     5:   invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
35     8:   return
36   LineNumberTable:
37     line 6: 0
38     line 7: 8
39 }
```

**Figure 2.11:** Compiled version of Hello World in bytecode mnemonics.

2. **Iterators simplify the aggregate interface:** it obviates the need for a similar interface in aggregate, thereby simplifying the aggregates interface.

3. **More than one traversal can be pending on an aggregate:** an iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Iterators are common in object-oriented systems. Most collection class libraries provide iteration abstraction by means of the Iterator design pattern. Figure 2.12 illustrates how the Java programming language implements the iterator design pattern.

The iterator interface defines a `hasNext()` method which can be used to check if there are any elements left to traverse. The `next()` method returns the current element in the data

```
1  interface Iterator<E> {
2   boolean hasNext();
3   E next();
4   void remove();
5  }
```

**Figure 2.12:** Iterator design pattern in Java.

structure and as a side effect moves the cursor to the next element. Finally, the `remove()` method removes the last element returned by the iteration. Additionally, there is an interface `Iterable<T>` which marks any data structure as supporting traversal of its elements by implementing the `Iterator<E>` interface. This is convenient as it can be used in Javas enhanced `for` loop construct to traverse the elements of a data structure implicitly. Figure 2.13 demonstrates an example illustrating this idea. In the example there is a `Tree<T>` data structure, in this case holding `Node<T>` structure. Since `Tree<T>` implements the `Iterable<T>` interface, the enhanced for loop construct knows that it can call the `hasNext()` and `next()` methods to traverse through the elements of the tree.

## 2.3 Generators

A generator is a special routine that can be used to control the iteration behaviour of a loop. It supports the Iterator design pattern, described in the last section. Generators are not something new, they can be traced back to the 70's in languages like CLU [32] and Icon [39]. Generators have found their way back into new programming languages like Python, Ruby and C# as a way to provide easy implementation of iteration abstraction and traversal of elements in data structures.

```
1  public class Node<T> {
2   // code omitted
3  }
4
5  public class Tree<T> implements Iterable<T> {
6   // code omitted
7  }
8
9  Tree<Node> tree = new Tree<Node>();
10
11 ... // the tree data is built up
12
13 for(Node n: tree) {
14  System.out.println(n);
15 }
```

**Figure 2.13:** Example using iterators in Java.

```
1  def range (begin, end):
2    while begin <= end :
3      yield begin
4      begin += 1
5
6  for x in range (1, 100)
7    print x
```

**Figure 2.14:** Generator example in Python.

```
1  public class CityCollection: IEnumerable<string>
2  {
3      string[] m_Cities = {"New York", "Paris", "London"};
4
5      IEnumerator<string> GetEnumerator()
6      {
7          for (int i = 0; i < m_Cities.length; i++)
8          {
9              yield return m_Cities[i];
10         }
11     }
12
13     IEnumerator<string> IEnumerable.GetEnumerator()
14     {
15         return GetEnumerator();
16     }
17 }
```

**Figure 2.15:** Generator example in C#.

A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array of containing all the values and returning them all at once, a generator yields the values one at a time, which requires less memory and allows the caller to get started processing the first values immediately.

Generators are usually invoked inside a loop. Upon its first invocation, an iterator object is created that encapsulates the state of the generator routine at its beginning, with arguments bound to the corresponding parameters. The generators body is then executed and the context of that iterator until a special yield action is encountered, at that time, the value provided with the yield action is used as the value of the invocation expression. The next time the same generator invocation is reached in a subsequent iteration, the execution of the generators body is resumed after the yield action, until yet another yield action is encountered. In addition to the yield action, execution of the generator body can also be terminated by a finish action, at which the innermost loop enclosing the generator invocation is terminated.

This idea is illustrated in Figure 2.14. The `range` function produces integers ranging from `begin` up to and including `end`. The `for` loop calls the `range` function much like an

iterator, but instead the `range` function yields an integer value and returns control to the `for` loop. This `for` loop binds the yielded value to the variable `x`. The next time the function `range` is called, it will continue with the statement following the `yield begin` statement. Figure 2.15 shows an example of C# generators (called iterators in C#).

# Chapter 3

# Program Transformation Techniques and Systems

In addition to the preliminaries, program transformation techniques and systems need be to discussed in order to understand the choices made for the implemented solution. Since this is an extensive topic, we have devoted an entire chapter for it. In this chapter the following techniques will be discussed: assimilating language extensions, open compiler frameworks, intermediate code transformation and Domain-Specific languages. All along with examples of systems that can be categorized into one or more techniques.

Program transformation is the process by which a program is changed, modified, altered into another with a certain objective: performance boost, specific generation of a program aiming at some specific application, extension of a language with some new convenient syntax, etc. In order to achieve this goal many techniques have been applied to generate code. The following sections will cover the different techniques found in literature that aim at assisting program transformation and the construction of transformation systems.

## 3.1 Assimilating Language Extensions

This is a technique that is often implemented in program transformation systems. It is based on rewriting new language constructs into the base language [20] and is more commonly known as source to source.

Language extension assimilation benefits from the ability to translate new language constructs into high level code, making it easier to understand its semantics. The disadvantages of this approach is that sometimes new constructs cannot be well ported onto the base language constructs, leading to a much grater effort to implement it or some aspects need to be traded off (performance, slightly change of semantics).

The Spoon framework [38] and OpenJava [40](now called OJ) are two program transformation systems for the Java programming language that can be categorized into this

technique. Both frameworks rely on compile-time reflection and build a model (sort of AST) of the source code to provide transformations. They both guarantee type safety (a requirement for Java programs), since they are type driven systems. The problem with both approaches is that they are limited by the technique they use, as earlier discussed in this section.

## 3.2 Open Compiler Frameworks

A programming language's compiler is the first suitable candidate when extending a programming language with a new construct. The problem with this approach is that it becomes a tedious job to implement the new extension. Small changes to the language syntax are reflected through the entire source code, since most compilers are not developed with extensibility in mind. Furthermore it requires a thorough understanding on compilers and its inner workings, knowledge which should not be necessary to extend a language with small features.

Another technique that gets around this problem is to use the existing compiler at the back end to generate the target code and use an open or extensible compiler framework at the front end. An open compiler framework makes it for the language extension implementer a much easier job by providing a set of tools to analyze the program structure, modify it and optimize it in order to support the language extension or program transformation in mind.

Polyglot [35] is an example of open compiler framework, JastAddJ [25] and the Dryad compiler [29] are examples of full extensible compilers. Polyglot benefits from the fact that, compilers implemented with Polyglot are extensible themselves. A disadvantage of this approach is that the system is limited to the degree to which a new language extension is portable to the Java language. JastAddJ and the Dryad Compiler are full compilers themselves. The problem with both is that they require to learn an specific language to implement any language extension. Also their power comes with a big deficiency: a big effort needs to be made in order to keep the system up to date with new enhancements and features of the Java programming language.

## 3.3 Intermediate Code Transformation

Some program transformation techniques focus on extending the compiler's front-end, since its quite difficult to get access to the compiler's back-end or the implementation could become a daunting task. Having access to the compiler's front-end has its limitation to what is possible. Therefore, research has focused on performing program transformation on the intermediate code produced by compilers for languages like Java and C#. This way one can surpass the compiler's back-end. Which leads to a whole range of possibilities for transforming the intermediate code.

The disadvantage of this approach is that, since it handles the low level aspects of the program, there is a price to pay in the additional complexity for implementing program transformations. The advantage is that it makes it possible to implement features that were not possible to translate into source code.

Focusing on the Java programing language, there are two frameworks that can be classified as program transformation systems, since they provide more or less the same functionality of extensible compilers, but for the bytecode instead of the high level Java source code. These two are ASM [22] and BCEL [23]. Both ASM and BCEL, deal with the problems of manipulating Java class files. The systems try to provide abstraction to the bytecode instructions, such that the programmer can focus on the transformation rather that the handling of bytecode instructions. The complexity to implement a transformation in the program is the trade of to be considered against the benefits gained by implementing the transformation.

### 3.3.1 The ASM Framework

In this section, we briefly discuss the ASM framework as it is the framework of choice for the implemented solution (see Section 4.3.3). ASM is a bytecode manipulation framework [21]. It has been designed to dynamically generate and manipulate Java classes. ASM does not mean anything, it is just a reference to the `asm` keyword in C, which allows some functions to be implemented in assembly language. The key idea of ASM is that it has been designed to be as fast as possible. It does not use an object representation of the class. Instead it uses the Visitor design pattern [26] but without explicitly representing the visited tree with objects. The Visitor design pattern can be used to manipulate the code of methods, and more generally the complete structure of a class, without creating one object per bytecode instruction. Finally, ASM allows to focus on code transformations instead of spending time on low level bytecode manipulations.

## 3.4 Domain-Specific Languages

A Domain-Specific language (DSL) is a small language developed to solve problems in well defined application domains. Van Deursen et. al has emphasized that many literature exist on Domain-Specific languages [41]. A key aspect of DSLs is that they are small and offer only a very restricted suite of notations and abstractions. Another aspect is that DSLs are usually declarative. This means that they can be viewed as specification languages, as well as programming languages.

Domain-Specific languages, in the context of program transformation, allow to define languages that specify transformations on programs. DSLs come in two flavours, external and internal[1]. External DSLs are what we have discussed so far as DSLs. These have the ability to express its domain as close as possible to the domain's terminology. It is only limited by the ability of the language designer to build a translator that can parse and produce

[1] In the literature internal DSLs are often called embedded DSL(see also [7])

an executable program, which is usually in a base language. Internal DSLs are based on the idea to inherit the infrastructure of some other programming language tailoring it such that it fits the domain of interest. The language is implemented by defining it semantics in an existing "host" language which constitutes a new language.

Defining an external DSL allows the definition of a language that fully capture the semantics of the intended transformations. The limitation lies in the ability to build a parser that is able to produce an executable program is some base language. On the other hand, internal DSLs, allow to make full use of the base language features, but fails to provide an optimal syntax for the intended domain and suffers from poor error messaging. The choice for a DSL approach, in the context of program transformation, depends on the requirements of the kind of transformations that are needed and the available infrastructure to perform these transformations. Stratego/XT [19], a program transformation system based on the Stratego transformation language and the XT tool set, is a good illustration of this. Its applicability depends on the willingness to learn the Stratego language and the usefulness of the tool in a targeted environment (i.e. Java source code in a commercial environment).

# Chapter 4

# Design Space

In the introduction section, the problem statement was clearly outlined. This chapter discusses the design space for this thesis project. It starts with the requirements for a suitable solution. This is followed by a discussion of attempts to port the generator construct into the Java programming language along with a motivation why it is not suited for our project. Finally, a clear discussion of the proposed solution is provided.

## 4.1 Requirements

Before designing and implementing a solution that will support the extension of the Java programming language with generators, it is essential to summarize the requirements in order to have a set of guidelines by which the extension can be measured against compliance upon completion.

### 4.1.1 An IDE Independent Solution

During the research of this thesis we found many solutions for other extensions that are very specific to an implementation and cannot be integrated within the existing tooling used in production environment such as modern existing IDEs (Integrated Development Environments). For instance, OpenJava [40] is a solution that requires the use of a preprocessor that would generate code for the base language. This solution would break the work flow in an IDE since the IDE editor would not recognize the new extension's syntax and hinder the developers from doing their work. A solution to this problem would be to implement a plugin for the specific IDE that would add support of the new language extension's syntax into the language. However, this would restrict developers from using other IDEs due to the dependencies created by this specific solution. Hence any solution to extend the Java programming language with generators should be deployable in any IDE without the need of IDE-specific plugins: These issues can be formulated as the following requirements:

**R1:** The implemented solution should be IDE-independent.

**R2:** The extension of generators in Java should be based on the existing language's syntax. Introduction of new syntax into the language is not allowed.

### 4.1.2 Java Compiler Independence

The Sun's Java compiler that comes with the JDK (Java Development Kit) has proven reliability through consistent development and testing which results into a wide acceptance between Java developers. It is therefore the industry standard. During the literature research, we came across solutions that required the use of alternate compilers. One of this solutions was the JastAddJ [25] compiler which supports new extensions to the Java programming language. The drawback of this approach is that it restricts the user to the usage of this compiler only. There is the concern as how fast and accurate new enhancements to the Java programming language will be implemented to comply with any modifications to the JLS and the and how the overall support of the compiler towards customers resembles Sun's Java compiler support. Therefore, the user should be allowed to have the freedom to choose the compiler that best fits his needs.

It is well known that some IDEs such as Eclipse use their own internal compiler (ECJ) to allow partial compilation as the code is being edited. It is an incremental Java compiler implemented as an Eclipse builder. In particular, it allows to run and debug code which still contains unresolved errors. However, for large scale projects that require a specific build process, these IDEs allow alternate compilation schemes such as ANT scripts. These alternate compilation schemes use mostly Sun's Java compiler. However other Java compilers can be used as well. Furthermore, these compilation schemes allow to build projects outside IDEs. It is under this setting how TOPdesk's application is built. Therefore we have the following requirement:

**R3:** A solution should be independent of any particular Java compiler.

Futhermore, no assumptions should be made about the settings[1] under which an application should be run in a JVM. Neither should an implementation making use of the generator construct require an application to run the JVM with any specific flags to provide its support. The reason for this is that we have no control as how the JVM is run and under which restrictions in a production environment. This results in the following requirement:

**R4:** Java programs making use of the generator construct should not require to run the JVM with any flags to provide its support.

### 4.1.3 Transparent Extension

As discussed in the previous sections, the solution should be Java compiler independent (R3), introduction of extensions to the new syntax is not allowed (R2) and the solution

[1] For instance, Sun's JVM can be run under special settings such as `-Xms<size` to set the initial Java heap size

should be IDE-independent (R1). Hence, the build process may be altered, provided that it produces the expected final results.

The solution for extending Java with generators should be transparent to the developer by integrating the solution into the build process. The developer need only to configure the build process once and use the extension as if it were native to Java without worrying about the implementation details. Furthermore, its semantics should be well defined. This can be formulated as the following requirements:

**R5:** The solution should be transparent to the developer such as that only an initial configuration is necessary (i.e. in the build process) without knowing the implementation details.

**R6:** The semantics of the generator construct should be well defined.

### 4.1.4 Debugging Support

The usage of a debugger facilitates the exploration of an application during run time. This exploration allows developers to examine the state of the code during its execution at any particular point of interest when the application goes through it. Nowadays, a debugger is of great importance in any software project as it aids in finding bugs, inconsistent states, and helps understand the behaviour of the application in general. Therefore, IDEs include support for debugging applications with many additional features such as visualization.

An introduction of a language extension such as generators must therefore provide debug support. A lack of debug support would impede developers from finding bugs and any inconsistent states since the developer would not be able to fully examine the application's source code state at the point where the extension is being used. In order to integrate the extension as transparent as possible, developers should be able to use the extension as if it were a native feature in the Java language. Hence, this includes the possibility to debug an application at the point where the extension is being used as well. This can be formulated as the following requirement:

**R7:** The generator language extension should provide debugging support.

### 4.1.5 Performance

The extension of Java with generators is intended to provide developers with an additional construct that facilitates an easier implementation of features in applications. One of the features is the traversal of data structures. A key factor in the success of the introduction of a generator construct into Java is the performance of the applications that make use of generators in their implementations. This can be formulated as the following requirement:

**R8:** The performance of an application implementing some feature by means of the generator construct should be at least the same as that one of an application implementing the same feature without its usage.

## 4.2 Existing Solutions

The Java programming language lacks support for generators. The language provides iteration abstraction only by means of the Iterator design pattern (see Section 2.2) and the `for` loop. Which is the main motivation of this project considering the benefits of generators. At the point of writing this document, there is no prospect in the Java Community Process requesting the implementation of generators for Java 7. This has not held back the Java community from attempting to provide some support for this construct. This section will provide a discussion of existing solutions for the Java programming language.

### 4.2.1 Asynchronous Implementation of Generators

Adrian Kuhn, a PhD student in computer science at the Software Composition Group, designed an asynchronous (threaded) implementation [13] of a generator based on the ideas of Ruby, Python and C#.

Figure 4.1 illustrates an example of the usage of this implementation. In the example, a generator is defined which produces the values of the Fibonacci series. The `run` method is overridden, which calls a `yield` method, so the value can be returned. The generator can be used in an enhanced `for` loop as depicted in the example. This implementation is based on an abstract class `Generator` which implements an iterator that yields its values one at at time. In order to make use of this functionality, one has to subclass the `Generator` class and override a `run` method which may call `yield(T)` to return values one at a time. At the heart of the implementation, there is an iterator object that runs in a separate thread. This thread is interrupted when new values are yielded. The generator that owns the thread waits on values produced by this iterator. According to the specification, an iteration terminates when a) a return statement is reached, b) the end of the generator method is reached, or c) when the generator's iterator gets garbage collected.

```
1  Generator<Integer> fibonacci = new Generator<Integer>() {
2   @Override
3   public void run() {
4    int a = 0, b = 1;
5    while(true) {
6     a = b + (b = a);
7     yield(a);
8    }
9   }
10 }
11
12 for (int x: fibonacci) {
13  if (x > 20000) break;
14  System.out.println(x);
15 }
```

**Figure 4.1:** Example of the usage of the asynchronous implementation of Generators in Java. Taken from [13].

```
1  class Fibonacci extends Yielder<Integer> {
2   public void yieldNextCore() {
3    int a = 0, b = 1;
4    while(true) {
5     a = b + (b = a);
6     yieldReturn(a);
7    }
8   }
9  }
10
11 for(Integer a: new Fibonacci()) {
12  System.out.println(a);
13 }
```

**Figure 4.2:** Example of the usage of Generators as supported by the Informancers Collection Library. Taken from [1]

This simplistic implementation works, but due to the threaded nature of its implementation, it suffers the performance of the threading mechanism inner workings. Recall that one of our requirements is acceptable performance (R8).

### 4.2.2 Informancers Collection Library

On Chaotic Java [1], there is an interesting article about a generator implementation for Java. A "yield return" like feature in Java as it is called in the article. The project started from the frustration of the blog's author of writing an implementation for a TreeModel. Soon, he realized that the implementation would have been much simpler if Java would a have a "yield return" like feature as in C#. In order to support this on top of Java, this functionality was implemented by means of a library [3].

Figure 4.2 shows an example of the usage of this implementation. Just like in the asynchronous example, the implementation of a generator is depicted which produces the values of the Fibonacci series. Here the user needs to extend a `Yielder` class, override a `yieldNextCore` method where the `yieldReturn` method must be used to return values. The technology used to implement the "yield return" like feature is based on bytecode manipulation and runtime instrumentation. The reason for this is that, there was no way to introduce a new keyword in Java and it was a requirement to be able to compile the code by a standard Java compiler. So when the class is loaded an agent is called by the JVM which is used to apply some rules over the loaded classes. The rules applied to the class are the manipulation of the bytecode in order to provide the implementation of the "yield return" like feature as in C#.

The library consists thus of an abstract class called `Yielder` and the agent used for the instrumentation. The class `Yielder` is `Iterable`. Which means that it can be used any-

where, where an `Iterator` would be needed. There is an abstract method `yieldNextCore` which needs to be implemented using the same style one would use to implement a .NET "yielding" method. In the method's body, one could call `yieldReturn(Object)` in order to return an item, or `yieldBreak()` to break the iteration loop. The semantics of this feature are as follows: every time the statement `yieldReturn(Object)` is called, one item is returned, the state of all local variables is kept, and finally at the next invocation of the method, the execution resumes after the last `yieldReturn(Object)` call. The agent provided by the library will be used by the JVM to instrument the loaded class whenever an instance of a Yielder class created in the code is found.

This approach is very practical and seems to work well. Still there are some remarks. First, in order to use this library it is required to run the JVM with the `-javaagent:yielder.jar` flag. This conflicts with our requirement of making no assumptions about the settings under which the JVM should run (R4). Secondly, this approach instruments the loaded class during runtime. This means that this instrumentation of bytecode manipulation happens at the time an instance of the Yielder class is loaded for the first time. This results in a slightly degradation of performance (R8). Thirdly, the implementation provides a specification of the semantics in a simplistic context, there is no specification about how this functionality would behave inside `try/catch/finally` blocks or within a threaded context. This goes against our requirement of well defined semantics (R6). Something that is crucial in order to provide a safe new feature, since the bytecode is manipulated and might lead to inconsistencies in the expected behaviour.

Finally, the library is to our knowledge neither widely used nor under acceptance of the Java community. Furthermore, this project is being maintained by a single developer and it is not certain how this project will be further developed.

### 4.2.3 Java Extension with the Dryad Compiler

The Dryad Compiler supports language extension and separate compilation of Java programs [29]. It uses a mixed Java/bytecode language as normalization step for language extensions to the Java programming language which is finally transformed into Java bytecode. As part of a demonstration of an application of the mixed Java/bytecode language for language extension, yield continuations (generators) were implemented.

Figure 4.3 shows how the language extension can be used. A generator is defined by having a method return an `Iterable` instance and have a `yield` statement in the body to return a value. The method can be used anywhere an `Iterable` instance can be used. As we can see from the example, this extension is similar in the way how generators are implemented in C# (see Figure 2.15) where a method must return an `IEnumerator` instance and have a `yield return` statement in the method body to return values. The difference in C# is that an `IEnumerator` is the equivalent to an Iterator in Java. The example depicted splits a given `String` based on spaces in multiple `String` tokens and returns each one at a time.

```
1  /**
2   * @return a word a a time at each invocation of the method.
3   */
4  public Iterable<String> getShortWords(String t) {
5   String[] parts = t.split(" ");
6   for (int i = 0; i < parts.length; i++) {
7    if (parts[i].length() < 4) {
8     yield parts[i];
9    }
10  }
11 }
```

**Figure 4.3:** Example of the generator language extension supported by the Dryad compiler. Taken from [29].

The approach taken to support this language extension was to treat the actual extension as a form of syntactic sugar. This form can then be projected to the base language (mixed Java/bytecode), just like it is done for the enhanced loop by the compiler since Java 5. The `yield` statement was implemented by rewriting the enclosing method to a small Java/bytecode class that implements the `Iterator` interface. The `yield` statement is rewritten to a return statement to exit the iterator method. At the beginning of the method a small jump table is introduced to allow the method to be resumed on re-entry. Based on a finite state machine model of the method, a state field is maintained that indicates the next method entry point. So the resulting program makes use of standard control flow statements in combination with unstructured goto instructions to be able to continue at multiple entry points. This way minimal changes are applied to the program so it can maintain its structure.

The example in Figure 4.4 shows the amount of code that would be required to implement the methods of the Iterator interface manually. The approach taken here uses for the implementation of the interface a central `prepareNext()` method that prepares the next element of the iterator. Which is done to implement the `next()` and `hasNext()` method (since assumptions cannot be made about the order they are called). Also, since it is not possible to determine statically if the iterator will return a value, given the state, a variable `valueReady` is used to indicate this during runtime.

The approach taken here to implement the support for generators seems to be simple and elegant. It tries to keep the structure of the original program by applying minimal changes. The drawback of this approach is that in order to benefit from the language extension the Dryad compiler must be used. This conflicts with our requirement of providing a compiler independent solution (R3). Another drawback is that a new syntax is introduced into language which conflicts with requirement (R2) that prohibits to do so. Therefore, this implementation cannot be used.

```
1  /**
2    * The transformed rewritten code to support the yield functionality.
3    */
4  class ShortWords implements Iterator<String> {
5   int _state = 0;
6   String _value;
7   boolean valueReady;
8   String _parts; // original code
9   int _i; // original code
10
11  private void prepareNext() {
12   if (valueReady || state == 2) return;
13   if (state == 1) `goto afterYield;
14
15   _parts = t.split(" "); // original code
16   for (i = 0; i < parts.length; i++) {
17    if (_parts[_i].length() < 4) { // original code
18     _state = 1;
19     _valueReady = true;
20     _value = =_part[_i]; // original code
21     return; // yield value
22     afterYield:
23    } // original code
24   } // original code
25   state = 2;
26  }
27
28  public String next() {
29   prepareNext();
30   if (!_valueReady)
31    throw new NoSuchElementException();
32   _valueReady = false;
33   return value;
34  }
35
36  public boolean hasNext() {
37   if (!_valueReady) prepareNext();
38   return _valueReady;
39  }
40
41  public void remove() {
42   throw new UnsupportedOperationException();
43  }
44 }
```

**Figure 4.4:** Example of the generator language extension internal transformation applied to the source code. Taken from [29].

## 4.3 Proposed Solution

In order to extend the Java programming language with generators, we proposed a solution based on trade offs from the requirements of generators and results of a previously conducted literature research. In this section, we provide an overview of the design of this solution which will be discussed in depth in Section 5.

```
1  public abstract class Generator<T> implements Iterable<T> {
2    public final Iterator<T> iterator() {/* code omitted */}
3
4    protected final void yieldReturn(T object) {/* code omitted */}
5
6    protected abstract void generate();
7  }
```

**Figure 4.5:** Proposed `Generator<T>` class.

### 4.3.1 Generator Support in Java Source Code

As depicted in Figure 4.5, each data structure that requires the usage of generators should extend an abstract `Generator<T>` class. This abstract class requires the implementation of the `generate()` method by client code to tell the iterator what to return on each call. Additionally, this class implements the `Iterable<T>` interface (so it can be used wherever an iterator can be used) and a `yieldReturn(T object)` method. The `yieldReturn(T object)` method should only be used inside the generator method. The call on the `yieldReturn(object)` method should return the corresponding object and return control to the client as explained in section 2.3.

Figure 4.6 shows an example of how the generator can be used. In the example the generator implements an algorithm for the Fibonacci series. This is similar to the usage of generators in the asynchronous implementation and the Informancers collection library.

The reason for having an abstract class is that this approach allows us to surpass the constraint of not being able to introduce a new syntax into the language, complying with requirement (R2). Hence, this approach allows us to define a "syntax" for generators without disturbing the current environment since it is defined in terms of the existing language's syntax. Furthermore, as this approach does not disturb the current environment, it can be

```
1  public class Fibonacci extends Generator<Integer> {
2    @Override
3    public void generate() {
4    int a = 0, b = 1;
5    while(true) {
6      a = b + (b = a);
7      yieldReturn(a);
8    }
9  }
10
11 for(int value: new Fibonacci()) {
12   System.out.println(value);
13 }
```

**Figure 4.6:** Example of generator usage as proposed in this thesis.

used with a compiler (R3) and IDE (R1) of choice.

### 4.3.2 Hybrid Approach

For our solution, we have chosen to base the support for generators on a hybrid approach. The reason for this is that it allows us to comply with requirement (R5), where the extension should be transparent to the user as if it were a Java native feature. This means that the build process is allowed to be altered and only an initial configuration should be required to use generators without knowledge of its internal implementation. However, we felt that in order to be non-intrusive, we need to support generators also for those environments where the build process in not allowed to be altered.

Therefore, by having a hybrid approach, support for generators can be transparently (R5) introduced in Java. This hybrid approach consists of two implementations that provide support for generators independently. The first implementation is the asynchronous implementation where the generator support is based on threads. The second implementation makes use of bytecode manipulation (bytecode weaving) to provide support for generators and is discussed in the next section. We mentioned earlier that the asynchronous implementation suffers from the performance of the threading mechanism inner workings. The choice is left to the developer who needs to make a trade off between altering the building process or accepting the performance degradation but leaving the build process intact.

We have chosen to provide as a standard solution the asynchronous implementation (see Section 4.2.1) since it is easily implemented in Java and could be integrated in the abstract `Generator` class. This implementation works independently from the bytecode weaving implementation and does not require any additional steps to use generators. It is this class the the user is required to extend. The inner workings of this class are discussed in depth in Section 5.1.1.

### 4.3.3 Generator Support at the Back-End

For environments that do allow changes in the build process, we apply the bytecode weaving approach (intermediate code transformation technique) to support generators. The reason for this is twofold. First, source to source transformation is limited by the ability to port the new language construct into the language. Since, generators require unstructured control flow constructs (i.e `GoTo` in combination with labels), we need access to the lower level code (bytecode) where this is possible as opposed to the high level constructs. The second reason is that, since we were not allowed to introduce any new syntax to the language (R2) and needed to introduce the language extension transparently (R5), we could not alter the Java code in any way that would break Java code as specified in the JLS. This would break the code and impede programmers from working with the tools (Java compiler, IDEs, etc) they require to do their job (R1, R3, R7).

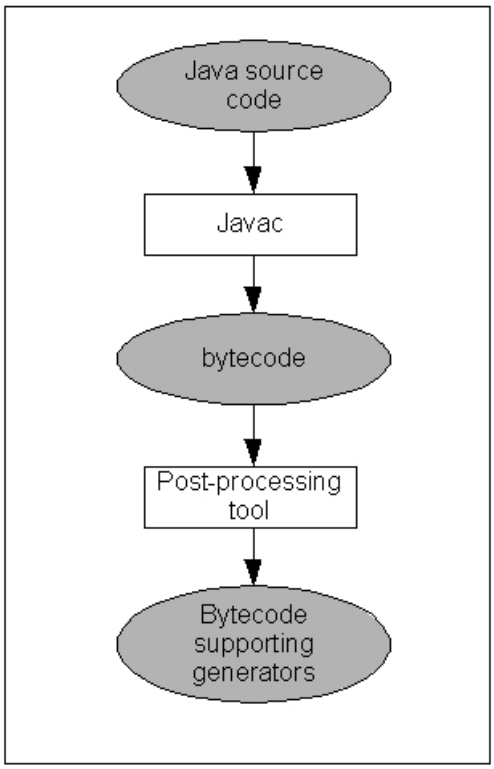


**Figure 4.7:** Proposal's Process Flow.

The `generate` method that needs to be overridden must be automatically integrated in the definition of the `iterator` method in the `Generator` class. This is where program transformations must be applied. We need to apply program transformation techniques in combination with a transformation system to provide the needed functionality on the back-end.

To support the bytecode weaving approach, we decided to use the ASM bytecode manipulation framework (see Section 3.3.1). The reason for this is that, ASM provides a high performance solution to manipulate Java classes. It solves many of the problems involved with manipulating Java classes satisfactorily and most importantly it allows the developer to focus on the program transformation logic instead of processing the bytecode instructions. Another reason was that it is the most widely accepted and used bytecode manipulation in many systems. It outperforms the BCEL, which is not longer being actively developed, but maintained for projects that once used it.

The application of bytecode weaving by means of manipulating bytecode through the use of the ASM framework gives us another advantage which is very important. The Java source code is compiled first after which bytecode is generated and stored in class files. This

bytecode is then post-processed by a post-processing tool (our bytecode program transformation tool) that recognizes the definition of generators and transform the program such that the needed functionality for generators is supported (see Figure 4.7). The advantage from this is that this kind of transformation is done at compile-time avoiding hurting performance and hence complying with the performance requirement (R8). Unlike other possible approaches such as bytecode instrumentation, classloaders, compiling new code by using the Java compiler API, which add to the performance complexity during runtime.

### 4.3.4 Generator Semantics

One of the requirements is to have well defined semantics for the generator construct (R6). The generator construct's semantics required two features that are not natively supported in the Java language:

- *Non-local control flow*: returning control back to the client code that makes use of the generator from an arbitrary `yieldReturn` call and resuming control from the next statement that follows the last `yieldReturn` call when the `generator` method is called again.
- *Preserving local variable state*: when a `yieldReturn` call is encountered, control is returned to the client code. Before returning control to the client code, the local state of all local variables should be preserved such that when control is resumed in the `generate` method, all variables can be used with their "current" values.

Exceptions are a non local control mechanism. Therefore, it is the only feature in the Java programming language that requires a clear definition of its behaviour inside generators. More specifically, we need to clearly define the expected interaction of `try/catch/finally` blocks with `yieldReturn` calls within the body of the `generate`. method. The behaviour of `throw` statements within the `generate` method need to be clearly defined as well. In this section the different cases are discussed.

#### The *Try* Block

In the `try` block there are two cases. The first case (see Figure 4.8) is when the `try` block contains a call to some method which can throw an exception and this method call is followed by a `yieldReturn` call.

The behaviour here is that when the `someMethod` call throws an exception, the exception handler is executed. Hence the body of the `catch` block is executed. On the next call of the `generate` method, the method executes from the beginning of its body since no `yieldReturn` call was done and hence no state needed to be preserved. If no exception is thrown, the `yieldReturn` call will be executed and on the next call of the `generate` method the method continues execution from the statement following the last `yieldReturn`

```
1  try {
2    Class.someMethod();
3    yieldReturn(value);
4  } catch (Exception e) {
5    // code omitted
6  }
```

**Figure 4.8:** Example of a `try` body with a `yieldReturn` call after a method call that can throw exceptions.

```
1  try {
2    yieldReturn(value);
3    Class.someMethod();
4  } catch (Exception e) {
5    // code omitted
6  }
```

**Figure 4.9:** Example of a `try` body with a `yieldReturn` call before a method that can throw exceptions.

call with the preserved local state.

The second case (see Figure 4.9) is when the `yieldReturn` method precedes a call to a method which may throw an exception. When the `yieldReturn` is called, control is return to the client code and the state of local variables is preserved. Upon resumption of the `generate` method, execution continues with the call of the `someMethod`. Should this call throw an exception, it is further handled as expected in the `catch` block and execution continues normally.

#### The *Catch* and *Finally* Blocks

Calls to `yieldReturn` are also allowed in the body of `catch` and `finally` blocks. The behaviour of `yieldReturn` calls in either of these blocks is the same. When a `yieldReturn` call is done (see Figure 4.10), control is returned to the client code and the local variable state is preserved[2]. Upon resumption of the `generate` method, execution continues with the statement following the last `yieldReturn` call.

#### The Throw Statement

Exceptions are thrown by means of the `throw` statement. Throw statements are allowed in the body of `generate` methods but only for unchecked exceptions. The reason for this is that in order to throw other checked exceptions, the overridden `generate` method would need to specify that it might throw these exceptions by means of the `throws` clause in its signature. Since the signature of the `generate` method in the `Generate` class does not

[2]In case of a `catch` body, the state of exception caught is preserved as well.

```
1  try {
2    // code omitted
3  } catch () {
4    ...
5    yieldReturn(something);
6    ...
7  } finally {
8    ...
9    yieldReturn(message);
10   ...
11 }
```

**Figure 4.10:** Example of `yieldReturn` calls in `catch` and `finally` blocks.

specify this[3], the overridden `generate` method must have the same signature without the `throws` clause.

There are two cases for which their behaviour in the `generate` method's body needs to be defined. The first case is when an unchecked exception is thrown and no `yieldReturn` call has been done yet (see Figure 4.11). In this case the `throw` statement would cause a disruption in the execution flow of the `generate` method. The execution stops and the exception thrown in handled at some point the program where it is specified. Upon resumption of the `generate` method. The execution starts at the beginning of the method since no `yieldReturn` call was done and hence neither the state of local variables was preserved nor a point in the method has been designated to run upon resumption. In the case where a `yieldReturn` call was done prior to the `throw` statement (see Figure 4.12), the `generate` method would resume from the statement following the last `yieldReturn` call. Finally, it is assumed that the code where the generator is being used (i.e. an enhanced `for`) handles the exceptions with `try/catch` blocks accordingly.

```
1  @Override
2  public void generate() {
3    ...
4
5    if (file == null) {
6      throw new NullPointerException();
7    }
8
9    yieldReturn(file.getPath());
10
11   ...
12 }
```

**Figure 4.11:** Example of a `throw` statement before a `yieldReturn` call in the body of a `generate` method.

[3]It was impossible to foresee which exceptions could thrown. Hence, we opted to have the user implementing the `generate` method handle the exceptions inside the method himself.

```
1  @Override
2  public void generate() {
3    ...
4
5    yieldReturn(file.getPath());
6
7    if (file.getParentFile() == null) {
8      throw new NullPointerException();
9    }
10
11   ...
12 }
```

**Figure 4.12:** Example of a `throw` statement after a `yieldReturn` call in the body of a `generate` method.

# Chapter 5

# Implementation

This chapter describes the architectural decisions and implementation of the post-processing tool that enables the support of generators in the Java programming language. It starts with a discussion of generator support at the front-end in terms of syntax and usage in the Java source code. This is followed by a discussion of the generator support at back-end by the post-processing tool. Then the transformation strategy to be applied by the post-processing tool is outlined and thoroughly explained. A discussion on the implementation of debugging support for generators is provided. Furthermore, we discuss the implications of transforming bytecode and some issues that arose while implementing the post-processing tool along with a discussion of the solution to these issues. Finally, this chapter is concluded with a discussion on the limitations of the implemented solution.

## 5.1 Generator Support in Java Source Code

```
1  public abstract class Generator<T> implements Iterable<T> {
2    public final Iterator<T> iterator() {/* code omitted */}
3
4    protected final void yieldReturn(T object) {/* code omitted */}
5
6    protected abstract void generate();
7  }
```

**Figure 5.1:** `Generator<T>` class structure.

As outlined in Section 4.3, we chose to introduce an abstract class `Generator<T>` (see Figure 5.1). The `Generator<T>` class implements the `Iterable<T>` interface such that it can be used anywhere an iterator is needed. This class requires to override a `generator` method where the traversal algorithm should be implemented. This implementation should contain a call to a `yieldReturn` method to which the generated value should be passed. This way we have a mechanism where generators can be implemented by overriding the `generate` method and calling `yieldReturn` in the implementation to produce values being

traversed without altering the syntax of the language (R2).

In our discussion of the proposed solution, we mentioned that two implementations of generators were to be supported. The first was the asynchronous implementation where the generator functionality was based on threads and the second where the functionality was based on transformed bytecode. To this end, there are two versions of the abstract `Generator` class. Both have the same structure as depicted in Figure 5.1. The difference lies in the implementation of the `iterator` and `yieldReturn` methods. Furthermore, it is the abstract `Generator` class based on threads that the user is required to extend. The other class is used by the post-processing tool as described in Section 5.2.

### 5.1.1 Anatomy of the Asynchronous Generator Class

The `Generator` class based on threads is implemented as follows. On call of the `iterator` method on a `Generator` instance an `iterator` object is returned. This `iterator` object is a thread which is owned by the `Generator` instance. The `iterator` thread calls the overridden `generate` method of the `Generator` instance. When a `yieldReturn` call is done in the `generate` method, a call is done on the `take` method which is defined in the abstract `Generator` class. This method produces a value by setting some state variables in the `Generate` instance and keeping record of the produced value. Since the `iterator` thread called this method, this thread notifies other threads and interrupts itself so that others threads can do their work. This is how the return of control and resumption is achieved in a generator. The `next` method checks if values have been produced, otherwise it tries to produce them by calling the `take` method. If no more values can be produced, it throws a `NoSuchElementException`. The `hasNext` method checks if there is a value and reports on this, otherwise it tries to produce it by calling the `take` method and report on this. Figure 5.2 illustrates this concept.

### 5.1.2 Anatomy of the Abstract Class Used by The Post-Processing Tool

The `Generator` class used by the post-processing tool is implemented as follows. On call of the `iterator` method on a `Generator` instance an `iterator` object is returned. The `iterator` object can be used by calling the `next` or `hasNext` methods. When the `next` method is called, it checks if an item has been produced. If this is not the case it tries to do so by calling a `prepareNext` method defined in the `Generate` instance. This method resets some state variables and calls the overridden `generate` method. When a `yieldReturn` call is done in the `generate` method some state variables are updated and the value produced is stored. This stored value is returned by the `next` method. If no value is available, the `next` method throws a `NoSuchElementException`. This implementation assumes that a `yieldReturn` call in the `generate` method causes a return from this method and that resumption of the `generate` method continues from the next statement after the last `yieldReturn` call. The `hasNext` method checks if an item has been produced and reports

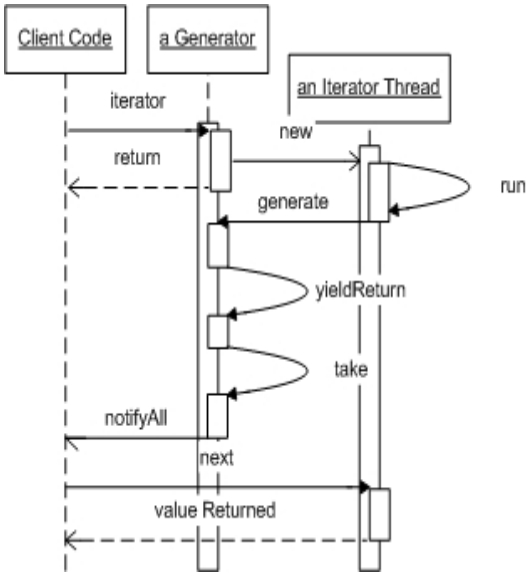


**Figure 5.2:** Sequence diagram of the `Generator` class used in the asynchronous approach.

on this, otherwise in tries to do so by calling `prepareNext` as already described and reports on this. Figure 5.3 illustrates this concept.

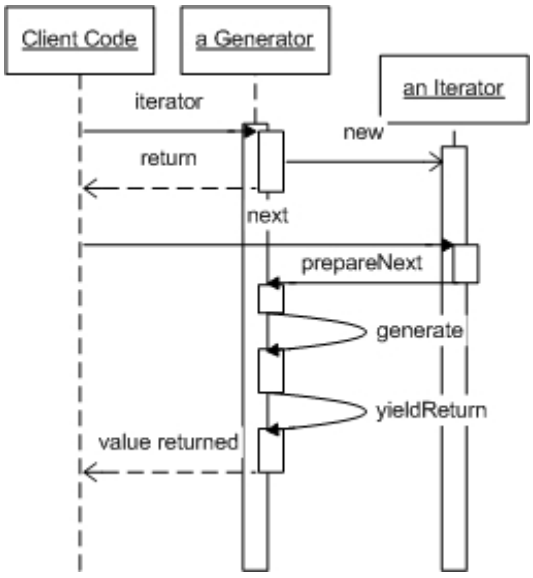


**Figure 5.3:** Sequence diagram of the `Generator` class used by the post-processing tool.

## 5.2 Generator Support at the Back-End

For those environments that do allow changes to the build process, there is another alternative. The build process is altered by having our implemented post-processing tool perform transformations on the compiled Java code. The post-processing tool changes the structure of the class that extends the `Generator` class such that it extends the alternate `Generator` class instead of the "standard" `Generator` class. This class does not rely on threads but on other implementation as explained in the previous section. This implementation assumes that the extended class will be transformed at the back-end by our post-processing tool in order to provide the generator functionality for the Java programming language.

This approach does not change the usage of generators in Java client code. It merely facilitates and simplifies the separation of both abstract `Generator<T>` class implementations and allows to the tool to recognize when a compiled class has already been transformed to support the generators. Figure 5.4 shows the process flow of the transformation as applied by out post-processing tool.

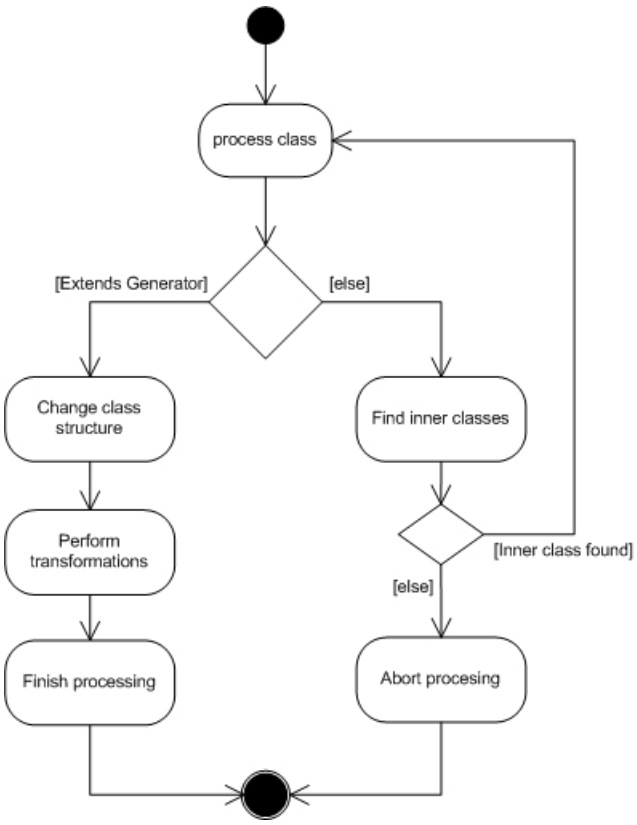


**Figure 5.4:** Back end process flow.

## 5.3 Applied Transformation Strategy

To port the generator construct into the Java programming language, we needed to perform transformations on the compiled code. This section describes the strategy applied and its steps in detail.

### 5.3.1 The Strategy

In order to provide non-local control flow and local variable state preservation we need to modify the compiled code (more precisely the class extending the `Generator` abstract class) as follows:

1. Introduce a `state` field to record the state of the control flow in the `generate` method. This allows to preserve state information across calls to the `generator` method.
2. Insert a lookup table at the beginning of the `generate` method that transfers control to the right position in the method according to the "recorded state" when the method is called.
3. Raise each local variable in the `generate` method to class level by introducing a new field in the class with the same type for each local variable.
4. If no `try/catch/finally` blocks are used in the `generate` method, save the local variables state into their fields variants before returning control to the client code and restore them after resumption of the `generate` method just before the instructions following `yieldReturn` call.
5. If `try/catch/finally` block are used in the `generate` method, replace each local variable reference in the method by a reference to its field variant. This will allow to preserve the value of local variables across calls to the `generator` method.

The reason for making a distinction between `generate` methods using `try/catch/finally` blocks and those which do not is as follows. As part of the exception handling mechanism of the JVM, the bytecode in a method contains a table with information on the exception being thrown at a certain point in the bytecode. Due to the transformations done in the bytecode, the class verifier in the JVM cannot provide guarantees about the state of local variables after `yieldReturn` calls in `try/catch/finally` blocks. The class loading process fails resulting in error messages from the JVM. Therefore only fields should be used to provide the guarantee of preserving local variables state allowing the class to load properly as it is succesfully verified by the JVM's class verifier.

Figure 5.5 shows a simplistic example where the `generate` method is implemented. This method contains two local variables `a` and `b` and two `yieldReturn` calls. Figure 5.6 shows how the body of the previous example would be transformed by applying this strategy. For simplicity's sake, we have depicted here a combination of Java source code and bytecode. In the following sections each step of the strategy will be explained in detail.

```
1  public void generate() {
2   int a = 0;
3   int b = 1;
4   yieldReturn(a);
5   a++;
6   yieldReturn(b);
7   b = a + b;
8  }
```

**Figure 5.5:** Simplistic implementation example of the `generate` method.

### 5.3.2 Introducing New Fields

In the previous section, we mentioned in our transformation strategy that we need to introduce new fields in order to support the generator construct's semantics. In the example depicted in Figure 5.6, we can see that the `generate` method contains two local variables `a` and `b`. For each of these variables, fields have been introduced. Furthermore, we can see how the state is recorded by copying the values to the fields and restored by copying these values back to the local variables.

Since all fields to be introduced will not be traceable to the source code, all of them need to be `private` so they can only be accessed from within the class. Furthermore, they need to be marked as `synthetic` as this attribute is used to mark class members that do not appear in source code. Finally, when introducing new fields, we have to make sure that no fields with the same name exist. For the sake of performance and ease of implementation we chose to precede each new introduced field with the prefix `_$`. This allows us to obviate the need to check for fields with the same name in the class being modified. Here we assume that any declared fields in the source code do not have this prefix.

The field to be introduced in all classes that, are being modified, is the `state` field. This field is declared in the class as follows:

```
private synthetic I _$state = -1;
```

We opted to use an integer field which is the type required for the lookup table (more on this in the next section). Furthermore, it has been initialized to -1 to mark the initial state.

As explained in section 2.1.3, java code is executed in threads containing its own execution stack. This stack is made of frames, each representing a method invocation. Furthermore each frame contains a local variables part. This part contains all the local variables and are accessed by an index. The local variable section is organized as a zero-based array. Any instruction using a value from or storing a value to a local variable must provide an index into the zero-based array. This has as consequence that there is no information on the local variables other than the size of the local variables part and the usage of the index by the bytecode instructions.

```
1  private synthetic int _state = -1;
2  private synthetic int _a;
3  private synthetic int _b;
4
5  public void generate() {
6   // lookup table
7   switch(_$state) {
8    case 1: goto L0;
9    case 2: goto L1:
10   default:
11    goto: L0:
12  }
13
14  L0:
15  int a = 0;
16  int b = 1;
17  yieldReturn(a);
18
19  // state is recorded
20  _$state = 1;
21  _$a = a;
22  _$b = b;
23  return;
24
25  L1:
26  // state is restored
27  a = _$a
28  b = _$b
29
30  a++;
31  yieldReturn(b);
32
33  // state is recorded
34  _$state = 2;
35  _$a = a;
36  _$b = b;
37  return;
38
39  L2:
40  // state is restored
41  a = _$a
42  b = _$b
43
44  b = a + b;
45 }
```

**Figure 5.6:** Java/bytecode source after transformation of the code of Figure 5.5.

To introduce any new fields that resemble all the local variables in the `generate` method, we need the type information (preferably their signature) first. Luckily for the primitives, it is easy to deduce their type, since bytecode instructions are typed, except for `boolean`, `short`, and `char` types (for these the type `int` is used). For any references (objects), we are able to infer their type as well from the bytecode instructions being used.

We are able to extract type information by looking at the `xLOAD` and `xSTORE` instructions.

Hence, whenever xLOAD *index* or xSTORE *index* is encountered, we introduce a field for the *index* if it has not already been introduced. The field is introduced with the type x if it is a primitive or inferred from the other instructions if it is an object. Since, we have no further information on the name of the local variable, we chose to use the prefix _$ along with the suffix localVariable*Index* where *Index* is the index of the local variable in question. For instance the assignment of 0 to local int variable in slot 1 in bytecode is as follows:

```
ICONST_0
ISTORE 1
```

The post processing tool will introduce an integer field (ISTORE so it is an integer) as follows:

```
private synthetic I _$localVariable1;
```

Furthermore the previous bytecode will be adapted as follows in the case of generate methods with try/catch/blocks:

```
ALOAD 0
ICONST_0
PUTFIELD <qualified class owner name>._$localVariable1 : I
```

Here the local variable reference is replaced by a field reference. ALOAD 0 will load this (the current object) into the operand stack. ICONST 0 loads the constant 0 into the operand stack. Finally PUTFIELD will store the value zero into the field _$localVariable1 (which was previously introduced) of the current object (this) on the operand stack. Hence, any xLOAD or xSTORE reference is replaced by a GETFIELD or PUTFIELD accordingly).

### 5.3.3 Inserting a Lookup Table

Inserting new fields resembling local variables in order to preserve local variable state allows us to comply with the local variable state preserving requirement. The other requirement (Non-local control flow) can be handled by inserting a lookup table that transfers control to the right position in the method according to the recorded state (see Figure 5.6).

The lookup table consists of a switch statement with as case statements for each possible state that allows to redirect control. Possible states are marked as *label* statements that precede the instructions that follow after a yieldReturn call (it is perfectly allowed to have multiple yieldReturn calls in the generate method). The example depicted in Figure 5.7 shows how labels work. The L1 statement is a label marking some point in a given set of bytecode instructions. In this case the ILOAD 2 instruction.

Switch statements in bytecode instructions are supported by two instructions TABLESWITCH and LOOKUPSWITCH. TABLESWITCH can be used when cases of the switch can be efficiently

```
1  ICONST_0  // Loads constant 0 into the operand stack
2  ICONST_1  // Loads constant 1 into the operand stack
3  IADD      // Pops the two constants, add them and puts the results back
4  ISTORE 2  // Stores the contents of the operand stack in local variable 2
5  L1:
6  ILOAD  2  // Puts the contents of local variable 2 on the operand stack
7  ISTORE 1  // Stores the contents of the operand stack into local variable 1
```

**Figure 5.7:** Example of labels in bytecode.

represented as indices into a table of target offsets. `LOOKUPSWITCH` can be used when the cases of the switch are sparse, hence the table representation becomes inefficient in terms of space. In our case we use `TABLESWITCH`, since we have continue values that can be represented as indices into a table.

We insert the lookup table at the beginning of the `generate` method in order to be able to transfer control to a certain point in the method (the statement following a `yieldReturn` call, according to the state or the "beginning" of the method if the state is initial). This allows us to mimic the resumption of the execution of the `generate` method from a random point in the code.

In Figure 5.6 we can see that a `switch`[1] statement has been inserted at the beginning of the method and makes use of the `_$state` field. Recall that we had two `yieldReturn` statements in the source code. Label `L1` designates the point after the first `yieldReturn` call and label `L2` designates the point after the second `yieldReturn` call. Additionally there is a third label `L0` which designates the start of the method (which is the point to jump to when the state is in the initial state) and hence the first line in source code[2].

Furthermore, we can also see that after each `yieldReturn` call, the `_$state` field is updated accordingly. This is followed by a `return` statement, which causes the method to return control to the client code. Finally, we can also see that all storing and restoring the state of local variables is achieved by copying the state to and from the equivalent fields respectively.

## 5.4 Debugging Support

The implemented solution is designed to be the non-intrusive. This means that supporting the semantics of the generator construct should not impede developers from being able to debug an application which uses generators (R7). Luckily, the Java virtual machine instruction set provides debug support as well (see Section 2.1.4). This section discusses how debug support for generators was implemented. It should be noted that debug support over-

[1] The corresponding bytecode instruction would be `TABLESWITCH`.

[2] Note that the `default` statement is not really needed here, but is put in the example to resemble the bytecode. Where the `TABLESWITCH` instruction has a default part

head is introduced by the post-processing tool only when debug information is available in the bytecode and the tool has been instructed to do so by means of the `-g` flag[3].

Supporting generators in the Java programming language involves a set of transformations. Fields are introduced resembling the local variables in the `generate` method. All references to local variables are replaced by their fields variants when `try/catch/finally` blocks are used in the `generate` method. Otherwise local variable state is stored in fields before a `yieldReturn` call and restored upon resumption of the method at the statement following this call. Finally, a lookup table is inserted to mimic the behaviour of arbitrary resumption in the `generate` method. Since local variables are used for the case when no `try/catch/blocks` are used in the `generate` method, no additional steps need to be taken to support debugging. For the opposite case, the transformations applied break the debug support, hence additional transformations need to be applied.

Since fields are used instead of the local variables in the `generate` method in the presence of `try/catch/finally` blocks, this means that the local variables will not have current value while being viewed in debug mode. To remedy this, when the post-processing tool detects debug information it will store the value of a field in the local variable slot whenever a value is stored to the field. This is done by duplicating whatever value is on the operand stack, after which one value is used to store it into the field and the other into the corresponding local variable slot.

Figure 5.8 illustrates this example. This figure shows the bytecode instructions for the `a++` statement in a `generate` method with `try/catch/finally` blocks, where the `_$localVariable1` field corresponds to a local variable `a` of type `int`. It also shows how our post-processing tool inserts extra instructions to reflect this changes on the actual local variable, so when it is viewed in debug mode, it will contain the current value. The `DUP` instruction was inserted after the `IADD` to duplicate the result on the operand stack and the `ISTORE 1` instruction was inserted after `PUTFIELD` instruction so the remaining value on the top of the stack is stored into the local variable slot 1. Which is the local variable `a`.

Duplicating values on the stack and storing the values on the local variables as well requires additional bytecode instructions might introduce some performance overhead. We feel that this is an acceptable trade off for adding debug support. Furthermore, it is only for the debug mode that this extra instructions are inserted. Hence, this performance overhead will not be applicable to production code (without debug information)[4].

The mapping of line numbers in source code to the corresponding set of bytecode instructions is done in the bytecode by maintaining a line number table (see Section 2.1.4) that maps line numbers to labels in bytecode. These labels group a set of instructions as be-

[3]The `-g:none` flag ignores debug information if available and hence the resulting bytecode has no debug overhead.

[4]It should be noted that the post-processing tool can be run with parameters to generate or ignore debug information. Furthermore, no extra operations are done if no debugging data is available.

```
1  L1 :
2  // a++
3  ALOAD 0
4  GETFIELD <qualified class name>.$localVariable1 : I
5  LDC 1
6  IADD
7  ALOAD 0
8  SWAP
9  PUTFIELD <qualified class name>.$localVariable1 : I
10
11 becomes
12
13 L1 :
14 // a++
15 ALOAD 0
16 GETFIELD <qualified class name>.$localVariable1 : I
17 LDC 1
18 IADD
19 DUP
20 ALOAD 0
21 SWAP
22 PUTFIELD <qualified class name>.$localVariable1 : I
23 ISTORE 1
```

**Figure 5.8:** Example of debug support transformation by the post-processing tool.

longing to the corresponding line number in source code. The post-processing tool makes sure that this line number table is properly maintained. Another issue remains due to the transformations performed by the post-processing tool. Inserting a lookup table at the beginning of the `generate` method means that the bytecode instructions being executed no longer correspond with the line number in the source code. The label defining the first line number and therefore the bytecode instructions belonging to the set of the first line number is incomplete. The instructions for the lookup table are now also part of this first line. This issue is easily fixed, by moving this label to the beginning of the instructions for the lookup table. Figure 5.9 illustrates this idea. Moreover, recall that the lookup table uses a switch statement with a label to the set of instructions corresponding to the first line in source code as a default target. This means that we still have a label designating the real set of instructions belonging to the first line in source code.

## 5.5 Implications of Bytecode Transformation

Performing transformations on the bytecode, as specified in Section 5.3.1 in order to support generators, has raised issues during the implementation due to the class file format structure and the bytecode instructions.

These issues surfaced while implementing the introduction of fields resembling local variables in the `generate` method. As already mentioned, the compiled code provides no information on the local variables. It is true that this information can be found in the local

```
1  L0:
2  // a = 0
3  ICONST_0
4  ISTORE 1
5  L1:
6  // b = 1
7  ICONST_1
8  ISTORE 2
9  ...
10
11 becomes
12
13 L0:
14 ALOAD 0
15 GETFIELD <qualified class name>._$state : I
16 TABLESWITCH
17   1:L1
18   2:L2
19   DEFAULT:L3
20 L3:
21 // a = 0
22 ALOAD 0
23 ICONST_0
24 PUTFIELD <qualified class name>._$localVariable1 : I
25 L4:
26 // b = 1
27 ALOAD 0
28 ICONST_1
29 PUTFIELD <qualified class name>._$localVariable2 : I
30 ...
```

**Figure 5.9:** Moving labels in bytecode to resemble the first line in source code.

variable table, but we cannot rely on this, since this is only meant for debugging purposes. The problem that we encountered was when local variable slots were being used for multiple local variables due to compiler optimizations or local variables being reused and instantiated as other types.

Figure 5.10 illustrates the case when primitive typed local variables share the same slot. In this example, the slots for local variables `i` and `j` at line numbers 7 and 8 are reused for local variables `k` and `l` at line numbers 14 and 15. This problem was solved by introducing only one field for each slot in the case that the local variables share the same type. In the example, the local variables `i` and `k` would share the same field and `j` and `verb` would share the same field. For the case when local variables share the same slot but have different primitive types, a field is introduced for each of them. For the remaining case of local variables sharing the same slot but have different non-primitive types (objects), only one field is introduced with the common super type of both variables. As a consequence of the solution for this case, the extra instruction `CHECKCAST` *`type`* needs to be inserted wherever this field (with the common super type) is used. The reason of this is that the casting preserves the type safeness as required in the Java language and checked by the bytecode verifier. The *`type`* parameter is here the type to which the instance will be cast.

```
1  @Override
2  public void generate() {
3    int[][] matrix = new int[SIZE][SIZE];
4
5    int value = 1;
6
7    for (int i = 0; i < matrix.length; i++) {
8      for (int j = 0; j < matrix[i].length; j++) {
9        matrix[i][j] = i + j + value;
10     }
11     value++;
12   }
13
14   for (int k = 0; k < matrix.length; k++) {
15     for (int l = 0; l < matrix[k].length; l++) {
16       yieldReturn(matrix[l][k]);
17     }
18   }
19 }
```

**Figure 5.10:** Example were slots are reused for multiple variable.

Figure 5.11 illustrates the case when local variables are instantiated as multiple types. The variable `list` is instantiated as an `ArrayList` and later on as a `LinkedList`. Recall that in bytecode we only have information about the instantiated types and hence we need to know which type the field to be introduced for this variable will have. There is no information specifying that the variable list is of type `List`. We can find the type in the debug information, but we cannot rely on this as it is not always available. Hence, this problem is solved by introducing a field with the common super type of both instances. The common super type here is `AbstractList`. Again, this solution requires the extra instruction `CHECKCAST` *`type`* to be inserted wherever this field is used. In the example, the statements in bold show where in the code the `CHECKCAST` instructions would be inserted. When debug information is available, the issue of local variables being instantiated as multiple types is not a problem. The type is simply obtained from the debug information.

## 5.6 Limitations

As with all approaches, there are limitations to the implemented solution. In this section we discuss these limitations.

### 5.6.1 Requiring Classes to Extend the Generator Class

Requiring a class to extend the `Generator` class would impede the class itself from inheriting from other classes. We do not believe that requiring a class to extend the `Generator` class would pose to many restrictions. The reason for this is that, iteration is often required for data structures. Data structures provide this by implementing the `Iterable` interface. With our approach the base class can extend the `Generator` class and all its subclasses need

```
1
2  @Override
3  public void generate() {
4    list = new ArrayList<String>();
5    checkcast ArrayList
6    list.d("T");
7    list.add("O");
8    for (String value: list) {
9      yieldReturn(value);
10   }
11
12   list = new LinkedList<String>();
13   checkcast LinkedList
14   list.d("K");
15   list.add("E");
16   for (String value: list) {
17     yieldReturn(value);
18   }
19 }
```

**Figure 5.11:** Example of a variable instantiated as multiple instances.

only to override the `generate` method. Another approach would be to create an inner class that extends the `Generator` class and override the `generate` method. This way the class can still inherit from other classes and provide iteration abstraction by using generators. This approach is common in software engineering practices where composition[5] and delegation[6] is heavily used in object-oriented programming languages to deal with this kind of issues. The following examples illustrate these ideas:

1. Class `BaseClass` extends `Generator` such that subclasses need only to override the `generate()` method (see Figure 5.12)

2. A class that needs to inherit from other classes could implement the `Iterable` interface and create an inner class where the method `generate` is overridden. One could call the `iterate` method of the inner class from the iterate method (see Figure 5.13).

### 5.6.2 Throwing Runtime Exceptions Only from the Generate Method

In Section 4.3.4, we discussed the behaviour of `throw` statements within the `generate` method. Furthermore, we also mentioned that only runtime exceptions can be thrown from a `generate` method.

The reason for this is that, throwing checked exceptions require the overridden `generate` method to specify that it might throw these. This is done in the signature of the method by

[5]Relationship in OOP between a class A and B where class A"has a" class B. As opposed to inheritance where class B "is a" class B.

[6]Design pattern for handling a task over to another part of the program. In OOP an object defers the task to another object, known as a delegate (see [26])

```
1  public class BaseClass extends Generator<Integer> {
2      protected List<Integer> list = new ArrayList<Integer>();
3
4      @Override
5      protected void generate() {
6          for (Integer value: list) {
7              yieldReturn(value);
8          }
9      }
10 }
11
12 public class SubClass extends BaseClass {
13     @Override
14     protected void generate() {
15         for (Integer value: list) {
16             yieldReturn(value * value);
17         }
18     }
19 }
```

**Figure 5.12:** Example using the Generator class in a class hierarchy.

```
1  public class BaseClass extends SomeClass {
2      protected List<Integer> list = new ArrayList<Integer>();
3  }
4
5  public class SubClass extends BaseClass implements Iterable<Integer> {
6
7      public Iterator<Integer> iterator() {
8          final Generator<Integer> doubler = new Generator<Integer>() {
9              @Override
10             public void generate() {
11                 for (Integer value: list) {
12                     yieldReturn(value * 2);
13                 }
14             }
15         };
16         return doubler.iterator();
17     }
18 }
```

**Figure 5.13:** Example using the Generator class without the need to subclass it.

means of the `throws` clause. The Java programming language allows to remove this clause in an overridden method or modify the `throws` clause such that it specifies that the method might throw a subclass of the specified checked exception in the original signature of the `generate` method. Moreover, it does not allow to introduce a `throws` clause in the signature of an overridden method whose original signature specifies none. Since the signature of the `generate` method does not specify a `throws` clause, the overridden `generate` method cannot introduce one. As runtime exceptions do not require to be specified in the signature of a method, these are the only exceptions that can be thrown from the `generate` method.

This has the limitation that the implementor of a `generate` needs to handle any checked exceptions in the method itself. We chose for this approach as we could not foresee the exceptions that could be thrown in the overridden `generate` method. We could have specified the `throws Throwable` clause in the signature of the `generate` method in the abstract `Generator` class. This would allow the implementor of the `generate` method to specify that this method might throw more specific exceptions since the Java programming language allows to specify subclasses of the `Throwable` class and all exceptions are descendants of the `Throwable` class. The problem with this solution is that, we would face the same restrictions in the abstract `Generator` class as the `generate` method is used in the `Iterator` object returned by a `Generator` instance (see Section 5.1). Hence, any exception thrown by the `generate` method would need to be handled here which has no use for the implementor of the generate method. Therefore, we left the handling of exceptions thrown in the `generate` method to be implemented by the user in the method itself.

# Chapter 6

# Evaluation

In order to evaluate the implemented solution in this thesis project, the following steps were taken to make sure that the implemented solution complies with the requirements discussed in Section 4.1. The functionality of the post-processing tool that performs the transformations was tested to have certainty that the tool performs the transformations correctly. Furthermore, we assessed the performance of the tool to have an idea of the introduced overhead in the compilation process. Then, the source code of TOPdesk's application was examined to give an estimate where the code could benefit from generators. Furthermore, a case study was conducted to assess applications of generators and their performance. Finally, a comparison was drawn between the Informancers collection library implementation and our solution. This chapter will discuss each of these steps in detail.

## 6.1 Evaluating the Post-Processing Tool

The first step in the of the implemented solution is testing the functionality of the post-processing tool. This means that the transformation applied to class files to support the generator construct, must be tested for compliance. The functionality was tested by means of unit tests that cover all of the aspects of the applied transformation strategy. For the implementation of the unit tests, the JUnit test framework [17] was employed. These tests were not only run for the post-proccessed bytecode, but also for the implementation of generators making use of the asynchronous generator approach.

It is important to emphasize that the tests concern the functionality of the post-processing tool in terms of required input, output and expected behaviour of the resulting bytecode. An alternate approach could be where the ASM framework would be used to generate bytecode streams representing a class to be transformed such that it would support generators and test whether the post-processing tool perform the bytecode tranformations correctly. This would required to test the bytecode produced by the post-processing tool for certain expected bytecode patterns. This alternate approach would be a tedious and error prone job to implement and still does not give us the certainty that the produced bytecode behaves as expected since only its expected structure is tested for compliance and not the expected behaviour of the

produced bytecode. Therefore, our approach focuses on testing the behaviour of produced bytecode, taking into acount all the aspects of the transformation strategy.

The remainder of this section dicusses the different aspects in detail including the performance of the post-processing tool and how this relates to the overall compilation complexity of a Java source file.

### 6.1.1 Testing Primitive Typed Variables Lifting

One of the steps of the transformation strategy was to lift local variables to class level by introducing fields for each of them. Since in bytecode primitive types (see Section 2.1.4) have their own associated `xLOAD` and `xSTORE` instructions and are handled differently by the post-processing tool than non-primitive typed local variables[1], we have tested this lifting separately. Figure 6.1 shows an excerpt of a JUnit test for the lifting of a local `int` primitive typed variable. As we can see from this test, an inner `Generator` class is defined which implements the behaviour of generating a series of integers in a certain range. In this test the expected range of integers is generated independently and stored in an array. The actual values are stored in another array by having a generator produce the values. Finally, the arrays containing the expected and actual values are tested ensure they contain the exact same values with the `assertEquals` call from the JUnit framework.

This test allows to verify that the `int` primitive typed local variables `to`, `from` and `i` in the `generate` method are properly lifted. The local variable `to` is used to initialize the `i` variable in the `for` loop. The local variable `from` is used in the boolean test of the `for` loop and the `i` variable is increased in each iteration of the loop and is the value returned by the generator. Furthermore, we test also implicitly that the state preservation of these local variables is handled properly since on each call of the `generate` method, the method resumes execution after the `yieldReturn` call and hence the local variables must contain their previous state in order for the code in the generate method to exectue properly. Finally, we implicitly test that the inserted switch statement along with its code is working properly. We deduce in fact that the aforementioned issues are handled correctly by the post-processing tool since the behaviour of the produced bytecode is as expected. It produces namely the correct values as instructed in the `generate` method and the test runs successfully. All other primitive types where tested in a similar manner.

### 6.1.2 Testing Non-Primitive Typed Variables Lifting

We mentioned earlier that the post-processing tool handles non-primitve typed local variables (called objects or references) differently. The reason for this is that objects are retrieved and stored through the `ALOAD` and `ASTORE` instructions respectively in bytecode. Furthermore, some source level static information is lost which leads to several issues that

[1]Non-primitve variables, also called objects or references, are handled in bytecode by the `ALOAD` and `ASTORE` instructions. This poses several challenges which are discussed in the following subsection.

```
1  @Test
2  public void intPrimitiveLocalVariableTest() {
3   final int FROM = 1, TO = 10;
4
5   Generator<Integer> range = new Generator<Integer>() {
6    @Override
7    protected void generate() {
8     int from = FROM, to = TO; // lifting up of this variables is tested
9     for (int i = from; i <= to; i++) {
10     yieldReturn(i);
11    }
12   }
13  };
14
15  // Generate expected values FROM upto TO
16  int[] expectedValues = new int[TO];
17  for (int i = 0; i < expectedValues.length; i++) {
18   expectedValues[i] = i + 1;
19  }
20
21  // Gets the values from the generator
22  int index = 0;
23  int[]actualValues = new int[TO];
24  for (int actualValue: range) {
25   actualValues[index++] = actualValue;
26  }
27
28  //Compare the values
29  for (int i = 0; i < TO; i++) {
30   Assert.assertEquals(expectedValues[i], actualValues[i]);
31  }
32
33 }
```

**Figure 6.1:** JUnit test for the lifting of an `int` primitive typed local variable.

need to be solved in order to properly lift object variables and preserve their state.

We started testing the lifting of object variables by having a simple test with a local variable of type `String` in the `generate` method as show in Figure 6.2. Here, we test that the `String` typed local variable is properly lifted and its state is preserved across multiple calls of the generate method. The test retrieves values from the generator and checks this against a precomputed expected set of values for equalness. The setting of this test is similar to the one explained in the last subsection for primitive-typed local variables.

One of the issues that we found while lifting object variables is the following. Some variables are declared to be of some interface type and are instantiated to a type implementing this interface. This is how it is intended in Java. The problem with this is that this interface type cannot be deduced in bytecode since we are working only with the instantiated types in bytecode. For instance in Figure 6.3 we can see that there is a variable `list` of type `List`. It has been initialized as an `ArrayList`. We are only able to see in bytecode that the local variable is an `ArrayList`. This is not a problem if the variable is used only

```
1  @Test
2  public void stringLocalVariableTest() {
3   final String TOKEN = "TOKEN";
4
5   Generator<String> tokenGenerator = new Generator<String>() {
6    @Override
7    protected void generate() {
8     String token = TOKEN; // lifting up of this variable is tested
9     for (int i = 0; i < token.length(); i++) {
10     yieldReturn(String.valueOf(token.charAt(i)));
11    }
12   }
13  };
14
15  int index = 0;
16  String[] actualTokens = new String[TOKEN.length()];
17  for(String token: tokenGenerator) {
18   actualTokens[index++] = token;
19  }
20
21  for (int i = 0; i < TOKEN.length(); i++) {
22   assertEquals(String.valueOf(TOKEN.charAt(i)), actualTokens[i]);
23  }
24 }
```

**Figure 6.2:** JUnit test for the lifting of a local variable of type `String`.

as an `ArrayList` throughout the `generate` method. But what type should the lifted local variable `list` be at class level if it is further in the method assigned a `LinkedList` as in the example? The solution we opted for is to use the common super type of `ArrayList` and `LinkedList` in these cases. Hence a field of type `AbstractList` which will reference to an `ArrayList` instance and further in the method to a `LinkedList` instance. Furthermore, this example is a unit test for this issue. In the test, the values generated by the generator self are tested for equality, showing that the lifiting and state preservation of the local variable `list` is handled correcly by the post-processing tool.

Finally, there is a remaining issue of local variables being initialized to `null` (see Figure 6.4). This is a bad coding practice, nevertheless it must be dealt with since the post-processing tool must be able to handle all cases properly. The problem with this is that due to source level syntax information lost in bytecode we cannot know the type of the local variable at that point (`null` initialization). The solution to this problem is to initially create a field of type `Object` and later on when there is more information further in the bytecode deduce its type and change the type of the previously created field to the correct one. Figure 6.4 shows an excerpt of the `generate` method used in a Generator to test this issue. This `generate` implementation corresponds to a test that walks a tree in depth first search order where the values of the nodes are retrieved and expected in a certain order. The tree contains actually the alphabet letters in depth first order. We test that the values are generated correctly. The important key here to note is that the `currentNode` variable was initialized to `null` and is used to hold temporary values in the algorithm. Since the values are gener-

```
1  @Test
2  public void localVariableWithDifferentInstances() {
3   Generator<String> generator = new Generator<String>() {
4    @Override
5    protected void generate() {
6     // List is here first an ArrayList and then a LinkedList
7     // The common super type of both would be AbstractList
8     // The test is that a field must be created of type AbstractList
9     // and the two yield return calls.
10    List<String> list = new ArrayList<String>();
11    list.add("T");
12    list.add("O");
13    for (String value: list) {
14     yieldReturn(value);
15    }
16
17    list.add("Test");
18
19    list = new LinkedList<String>();
20    list.add("K");
21    list.add("E");
22    for (String value: list) {
23     yieldReturn(value);
24    }
25   }
26  };
27
28  String[] expectedValues = {"T","O","K","E"};
29  String[] actualValues = new String[expectedValues.length];
30  int index = 0;
31  for (String value: generator) {
32   actualValues[index++] = value;
33  }
34
35  for (int i = 0; i < expectedValues.length; i++) {
36   assertEquals(expectedValues[i], actualValues[i]);
37  }
38 }
```

**Figure 6.3:** JUnit test for an object local variable with a common super type.

ated correctly as expected and hence the test runs successfully, we conclude that the lifting and state preservation of variables with this issue is handled properly by the post-processing tool.

### 6.1.3 Testing Array Variables Lifting

In Section 2.1.4, we mentioned that for loading and storing values from and to arrays, the bytecode instruction set has instructions designed specifically for this purpose. It is for this reason that, despite the fact that arrays are Objects (references) in bytecode, we chose to test these separately in order to be certain of the proper lifting and state preservation of local array variables by the post-processing tool.

Local array variables can be tested against the following issues. Local array variables

```
1  @Override
2  public void generate() {
3   // Here we test that variable currentNode is
4   // handled well since it's been initialized as null
5   DefaultMutableTreeNode currentNode = null;
6   List<DefaultMutableTreeNode> visitCandidates =
7                       new ArrayList<DefaultMutableTreeNode>();
8   visitCandidates.add((DefaultMutableTreeNode)model.getRoot());
9
10  currentNode = visitCandidates.get(0);
11
12  while (currentNode != null) {
13   yieldReturn(currentNode);
14
15   // put left child in queue
16   if (currentNode.getChildCount() > 0) {
17    visitCandidates.add((DefaultMutableTreeNode)currentNode.getChildAt(0));
18   }
19   // put right child in queue
20   if (currentNode.getChildCount() > 1) {
21    visitCandidates.add((DefaultMutableTreeNode)currentNode.getChildAt(1));
22   }
23
24   // remove the parent
25   visitCandidates.remove(0);
26
27   if (visitCandidates.size() > 0) {
28    currentNode = visitCandidates.get(0);
29   } else {
30    currentNode = null;
31   }
32  }
33 }
```

**Figure 6.4:** JUnit test excerpt of a `generate` method used in a Generator to test the lifting of local variables that are initially initialized to `null`.

can be primitive typed and non-primitive typed. Recall that this is important since some source level syntax information is lost in bytecode. Furthermore, local array variables can be one-dimensional or multi-dimensional. Finally, array variables can be instantiated through the `new` *`type`* syntax or instantiated and initialized through the curly braces {} syntax.

Lifting primitive typed local array variables was tested as show in Figure 6.5. Here, a one-dimensional array of type `int` was instantiated and dynamically initialized. The expected values were tested for equalness against an expected precomputed set of values, leading to a successful run of the unit test. This shows that the post-processing tool can handle the lifting and state preservation of primitive typed array variables initialized through the `new` *`type`* syntax. The case for lifting non-primitive typed one-dimensional local array variables was handled similarly as shown in Figure 6.6. Here, lifting a one-dimensional array variable of type `String` is tested that is initialized through the curly braces {} syntax. Furthermore, we tested similarly primitive typed multi-dimensional local array variables and the non-primitive multi-dimensional variant as shown respectively in the excerpts of

```
1  @Override
2  protected void generate() {
3   // lifting array of primitive typed int is tested
4   int[] array = new int[SIZE];
5   for (int i = 0; i < array.length; i++) {
6    array[i] = i + 1;
7    yieldReturn(array[i]);
8   }
9  }
```

**Figure 6.5:** JUnit test excerpt of the `generate` method used in a generator for a primitive typed one-dimensional local array variable.

```
1  @Override
2  protected void generate() {
3   // lifting array of type String is tested
4   String[] stringArray = {"T", "O", "K", "E", "N"};
5   for(int i = 0; i < stringArray.length; i++) {
6    yieldReturn(stringArray[i]);
7   }
8  }
```

**Figure 6.6:** JUnit test excerpt of the `generate` method used in a generator for a non-primitive typed one-dimensional local array variable.

Figures 6.7 and 6.8. In Figure 6.8 we can see that the `generate` method declares an array of type `String` with four dimensions. It has been instantiated and initialized with the curly braces syntax containing several expressions. The first expression is the assignment to a `String` variable whose value is stored in the array. The second expression is the value returned by method whose value is also stored in the array. The result of the unit test was successful, ascertaining that the post-processing tool can handle the lifting and state preservation of local array variables in a proper manner.

```
1  @Override
2  protected void generate() {
3   // lifiting up of primitive int typed array is tested while
4   // being initialized directly
5   int[][] matrix = {{1, 2}, {3, 4}};
6
7   for (int i = 0; i < matrix.length; i++) {
8    for (int j = 0; j < matrix[i].length; j++) {
9     yieldReturn(matrix[j][i]);
10   }
11  }
12 }
```

**Figure 6.7:** JUnit test excerpt of the `generate` method used in a generator for a primitive typed multi-dimensional local array variable.

```
1  @Override
2  protected void generate() {
3   //This initialization is tested
4   String b;
5   String[][][][] matrix = {{{{b = "6"}, {returnString(), "2"}}}};
6
7   for (int i = 0; i < matrix.length; i++) {
8    for (int j = 0; j < matrix[i].length; j++) {
9     for (int k = 0; k < matrix[i][j].length; k++) {
10     for (int r = 0; r < matrix[i][j][k].length; r++) {
11      yieldReturn(matrix[i][j][k][r]);
12     }
13    }
14   }
15  }
16 }
```

**Figure 6.8:** JUnit test excerpt of the `generate` method used in a generator for a non-primitive typed multi-dimensional local array variable.

### 6.1.4 Testing Try/Catch/Finally Blocks within Generators

The Java programming language provides the `try/catch/finally` syntax for proper error handling in programs (see Section 2.1.4). Exceptions disrupt the normal flow of the program and more specifically of a method. It is a non local control mechanism. Therefore several cases have been tested to be certain of the correct behaviour in generators.

In each case an exception is thrown from the `generate` method implementation in order to test the behaviour of `try/catch/finally` blocks. The cases for which the use of `try/catch/finally` blocks has been tested are as follows. The first case is a generator having in its `generate` method a `try/catch` block and a `yieldReturn` call in the `try` section as shown in Figure 6.9. What we are testing here is that resumption in the `generate` method is possible within the `try` section and that throwing an exception does not disrupt the resumption of the `generate` method from a particular point as it is expected with generators. The corresponding JUnit test, tests that the generator values match a precomputed set of values for equality. The test runs successfully, indicating that the generator behaves as expected for this case.

The second case concerns calling `yieldReturn` inside the `catch` section of the `try/catch` block when an exception is being thrown as depicted in Figure 6.10. Here, we test that `yieldReturn` calls can be done from `catch` blocks. The JUnit test uses a generator with the implementation of the depicted `generate` method. A series of `String` values are generated and match for equalness against a precomputed expected set. The series of values depend on values that are generated from the two `catch` sections in the `generate` method. These are run interchangeably as the `generateException` method is used in the `try` section and designed to alternate the thrown exceptions. The test runs successfully, ascertaining that

```
1  @Override
2  protected void generate() {
3   //Tests that we can resume execution even when an exception
4   //has been thrown after a yieldReturn call.
5   while (i < LOOPS) {
6    try {
7     yieldReturn(i++);
8     generateException();
9    } catch (Exception e) {
10
11   }
12  }
13 }
14
15 private void generateException() throws RuntimeException {
16  throw new RuntimeException("Exception from a generator");
17 }
```

**Figure 6.9:** JUnit test excerpt of a generate method used by a generator to test a `yieldReturn` call from the `try` section in a `try/catch` block.

behaviour of the generator is as expected under the circumstances described by this case.

Finally, there is the case where `yieldReturn` calls are done in the `finally` section of a `try/catch/finally` block as shown in Figure 6.11. The `generate` method implemented here is used by a generator in a JUnit test. In the JUnit test the generator is used to generate a series of `String` values which are matched against a precomputed set of values for equalness. This set of values depend on the values generated by the `finally` section as well. The test runs successfully, ascertaining that the generator behaviour under this conditions is as expected.

### 6.1.5 Testing The Post-Processing Tool with/without Debugging Support

After testing the internal functionality of the post-processing tool in terms of the produced tranformed bytecode and expected behaviour of it, careful consideration was taken to other aspects as well. We tested the post-processing tool with class files containing debug information for all tests mentioned in the previous sections as well as with class files lacking debug information. The reason for this is that classes can be compiled with options to generate debug information or without. Having debug information is convenient since it will facilitate the post-processing tool with type information that is otherwise not deduceable from the bytecode only. By having all tests run under both conditions, we can be certain that the post-processing tool perform all the transformations properly under any circumstances.

```
1  @Override
2  protected void generate() {
3   //Tests that we can resume execution even when an exception
4   //has been thrown after a yieldReturn call and that yield return
5   //calls can be done in catch blocks
6   while (index < TOKEN.length()) {
7    try {
8     yieldReturn(String.valueOf(TOKEN.charAt(index++)));
9     generateException();
10   } catch (IllegalArgumentException e) {
11    e.getCause();
12    yieldReturn("*");
13   } catch (IllegalStateException e) {
14    e.getCause();
15    yieldReturn("-");
16    }
17   }
18  }
19
20  private void generateException() {
21   if (index % 2 == 0) {
22    throw new IllegalArgumentException("Some error");
23   } else {
24    throw new IllegalStateException("Some error");
25   }
26  }
```

**Figure 6.10:** JUnit test excerpt of a generate method used by a generator to test a `yieldReturn` call from the `catch` section in a `try/catch` block.

## 6.2 Performance of the Post-Processing Tool

In order to gain some information about overhead added to the compilation time of a Java source file and the post-processing of the generated class file, the performance of the post-processing tool was assessed. We assessed the performance by compiling and post processing a small class that only implements a generator and a big class that implements lots of generators as inner classes. We did the compilations and post-processing runs with and without generating debug information. We did the runs for each situation ten times. The average values from these results were taken. Figure 6.12 shows the results. All the values are in seconds. As we can see, compiling and post-processing while generating debug information takes more time for both cases. This is expected as both tools need to perform extra steps. Furthermore, the post-processing step is a fraction of the time needed to compile the class for both small and big classes. From the results, we observe that this is at least half of the compilation time. Hence, the compilation overhead added to the compilation time of a Java source file with the post-processing tool is a half more time.

```
1  @Override
2  protected void generate() {
3   //Tests that we can resume execution even when an exception
4   //has been thrown after a yieldReturn call and that yieldReturn
5   //calls can be done in finally blocks
6   int index = 0;
7   while (index < TOKEN.length()) {
8    try {
9     yieldReturn(String.valueOf(TOKEN.charAt(index++)));
10    generateException();
11   } catch (Exception e) {
12    // some code here
13   } finally {
14    yieldReturn("-");
15   }
16  }
17 }
18
19 private void generateException() {
20  throw new RuntimeException("Illegal state");
21 }
```

**Figure 6.11:** JUnit test excerpt of a generate method used by a generator to test a `yieldReturn` call from the `finally` section in a `try/catch/finally` block.

| | **Java compiler** | | **Post Processor** | |
|---|---|---|---|---|
| | **Debug Info** | **No Debug Info** | **Debug Info** | **No Debug Info** |
| **Small class** | 1.2 | 1.0 | 0.65 | 0.67 |
| **Big class** | 1.5 | 1.7 | 0.7 | 0.9 |

**Figure 6.12:** Performance measurements of the post-processing tool and the Java compiler. All the values are in seconds.

## 6.3 Identification of Translation Patterns

In order to show the applicability of the implemented solution, we examined TOPdesk's application source code in order to identify patterns that could be used to translate portions of the source code into generators. TOPdesk's code base has approximately 2MLOC. It consists of 39 projects which are the different modules of the TOPdesk application. We tried to identify as much patterns as possible that are closely related to `Generators` in the code base. Hence the translation of `Iterable` and `Iterator` instances into `Generators` at class and method level. We realize that there are other patterns and code implementations that can be identified as generators or benefit from the use of generators. These patterns are less obvious and more difficult to find. Given the complexity of the code base, we chose to look for other patterns and applications of generators by implementing small examples as it is discussed in the next section. This section describes the steps taken that led to the identification of some patterns.

### 6.3.1 Manual Identification

Since it was not clear what sort of patterns we were looking for, we started first with a manual search for the following pattern:

- Find in TOPdesk's source code all the classes that implement the `Iterable<T>` interface.

This manual search resulted in 65 classes implementing the `Iterable<T>` interface. We proceeded with a manual translation of the classes into our new introduced generator construct where it was possible. We managed to do 14 straight translations. We found out that most of the translations could be done easily by making the class, implementing the `Iterable<T>` interface, extend the `Generate<T>` class instead. The same goes for inner classes. For anonymous inner classes defining an anonymous inner `Iterable<T>` instance, the translation was made by defining an anonymous inner `Generator<T>` instance instead.

Furthermore, the translation internally proceeded by putting the `hasNext()` body in the boolean expression of a `while` statement in the `generate` method. The `remove` method could almost always be removed, since it was not supported[2]. The contents of the `next()` method could easily be adapted by putting its contents inside the body of the `while` loop of the `generate` method and removing any unnecessary temporary variables. Finally, some fields used in the `Iterable<T>` implementing class could also be moved to local level by demoting them to local variables. It is important to outline that some cases needed some careful interpretation while translating the `next()` method in the `generate` method, since each implementation is unique.

Figure 6.13 shows an excerpt of a class from TOPdesk's source code implementing the `Iterable<T>` interface while Figure 6.14 shows the code after the actual translation. Here, we have a class `SimpleResultSetBuilder` which defines an inner static class `SimpleResultSet` which implements the `Iterable` interface. The `SimpleResultSet` inner class is adapted to extend the `Generator<T>` class. This allows to remove the `iterator` method and replace it by the `generate` method. The implementation of methods of the anonymous `Iterator<RowData>` class was used in the `generate` method. A `while` loop was introduced with as boolean expression the returning statement of the `hasNext` method in the `generate`. The `remove` method was simply removed. The temporary variable of the `next` method was removed and the result stored by this variable was used instead in a `yieldReturn` statement, replacing the `return` statement. Finally, the field `currentRowNumber` was demoted to a local variable in the `generate` method.

The search for other patterns was conducted in a similar way as it has been described in this section. Section 6.4 provides an in depth discussion of the patterns found.

[2]The implementation of this method consists almost always of a `new UnsupportedOperationException()` statement.

```
1  public class SimpleResultSetBuilder {
2   ...
3   public static class SimpleResultSet implements Iterable<RowData> {
4    private final ResultSet resultSet;
5    ...
6    @Override
7    public Iterator<RowData> iterator() {
8     return new Iterator<RowData>() {
9      private int currentRowNumber = 0;
10
11     @Override
12     public boolean hasNext() {
13      return currentRowNumber + 1 < resultSet.rowCount();
14     }
15
16     @Override
17     public RowData next() {
18      final RowData row = resultSet.getRow(currentRowNumber);
19      currentRowNumber++;
20      return row;
21     }
22
23     @Override
24     public void remove() {
25      throw new UnsupportedOperationException();
26     }
27
28    };
29   }
30  }
31  ...
32 }
```

**Figure 6.13:** Example code of the implementation of the `Iterable<T>` interface in TOPdesk's source code.

### 6.3.2 Motivation for a Manual Identification

The identification of patterns has been conducted by means of a manual search and a manual translation of the source code. This raises the question of why this process was not automated?

The search process was in fact semi-automated. The search was performed by using the Eclipse IDE where queries were done on the source code. The queries were manually inserted and the results were used for further manual analysis. The queries were done by means of the *call* and *type* hierarchy options in Eclipse along with the Java search dialog. The reason for a manual search of patterns that could be translated into generators was a trade-off of the time needed to implement a fully automatic solution and the benefit gained within the scope of this project. In order to implement an automatic solution, the patterns needed to be identified first. This identification would require a manual search through the source code. After this identification, the implementation of an automatic solution would take place. This would be a tedious and error prone job that is outside of the scope of this

```
1  public class SimpleResultSetBuilder {
2   ...
3   public static class SimpleResultSet extends Generator<RowData> {
4    private final ResultSet resultSet;
5    ...
6    @Override
7    public void generate() {
8     int currentRowNumber = 0;
9     while (currentRowNumber + 1 < resultSet.rowCount()) {
10     yieldReturn(resultSet.getRow(currentRowNumber));
11     currentRowNumber++;
12    }
13   }
14  }
15  ...
16 }
```

**Figure 6.14:** Translation of the implemented `Iterable<T>` interface code in Figure 6.13 into the generator construct.

thesis project. Considering the focus of this thesis project on extending Java with generators, we chose to identify the patterns and document on the findings instead of investing time on the implementation of such an automatic tool.

## 6.4 Overview of Translation Patterns

After doing the manual search and translation into the generator construct, the following patterns and translation schemes can be identified at class level:

- A class or inner class implementing the `Iterable` interface can be adapted to extend the `Generator` class instead.
- A class implementing the `Iterable` interface and defining an anonymous inner `Iterable` instance in the `iterator()` method can be adapted to define an anonymous inner `Generator` instance in the `iterator()` method instead.
- A class having a method returning `Iterable` and defining an anonymous inner `Iterable` instance that is being returned can be adapted to define an anonymous inner `Generator` instance instead.
- A class having a method that returns an anonymous inner class which has a method returning an anonymous inner `Iterable` instance can be adapted by returning an anonymous inner `Generator` instance instead.
- An inner class having a method that returns an inner anonymous `Iterable` instance can be adapted to return an inner anonymous `Generator` instead.

- An interface with a field defining an anonymous interface instance with a method returning an anonymous inner `Iterable` instance can be adapted to return an anonymous inner `Generator` instead.
- A class that implements the `Iterable` interface having an inner class implementing the `Iterator` interface can be adapted by making this class extend `Generator` and removing the inner class. Provided that it is not used else where.
- An inner class that implements `Iterable` and returns an `Iterator` in the `iterator` method. This `iterator` is also an inner private class. One could extend `Generator` instead of implement `Iterable` and override the `generate` method instead of the `iterator` method. This means that the inner class can be deleted since it is private. It is namely only used in the method of the `Iterator`.
- An abstract class that implements `Iterable`, but does not implement the `iterator` method. It is left for the subclasses. The abstract class can extend `Generator` instead.

Figure 6.15 shows an example from TOPdesk's source code where one of the previously described patterns can be identified. This pattern describes a class that implements the `Iterable` interface and that has an inner class implementing the `Iterator` interface. The translation that can be applied is to adapt the class such that it extends `Generator` and remove the inner class when it is used only in the `iterator` method. Figure 6.16 shows the class after application of the advise for the translation pattern. As we can see, this results in a significance reduction of code and hence readability.

The following strategy can be applied to provide an advice of how to implement the `generate method`:

1. Introduce a `while` loop with as boolean expression the returning statement of the `hasNext` method.
2. Remove any unnecessary temporary variables holding a result that is being returned by the `next` method. Use the result directly and replace the `return` statement by a `yieldReturn` call with this result as argument. Place this `yieldReturn` call inside the previously introduced `while` loop with any other relevant code of the `next` method.
3. Demote any fields declared in the `Iterable` instance to local variable declarations. Initialize them here just as at class level. This should be done for fields that are only used in any of the methods specified by the `Iterator` interface.

In Figures 6.15 and 6.16, we showed an identified translation pattern and the application of the translation advised for this pattern. The strategy described for the advice to implement the `generate` method has been applied in this example as well. A `while` loop was introduced with as boolean expression the `i < list.getSize()` statement returned by the `hasNext` method (step 1). Any unnecessary temporary variables like the variable

```
1  package nl.ogdsoftware.tas.search;
2
3  public class LogicComponentList<T extends AbstractLogicComponent>
4  implements Iterable<T> {
5   ...
6
7   @Override
8   public Iterator<String> iterator() {
9    return new MyIterator;
10  }
11
12  private class MyIterator implements Iterator<T> {
13   private int i = 0;
14   private LogicComponentList list;
15
16   MyIterator(LogicComponentList list) {
17    this.list = list;
18   }
19
20   public boolean hasNext() {
21    return i < list.getSize();
22   }
23
24   public T next() {
25    T e = (T)list.get(i);
26    i++;
27    retrun e;
28   }
29
30   public void remove() {
31    throw new UnsupportedOperationException();
32   }
33
34  }
35
36  ...
37 }
```

**Figure 6.15:** Excerpt of a class from TOPdesk's source code where a pattern can be identified which is eligible for translation into a generator.

`e` in the `next` method have been removed. The `return` result in the `next` method is used directly and replaced by a `yieldReturn` call. This `yieldReturn` call is placed in the body of the `while` loop along with relevant code like the `i++` statement (step 2). Finally, the int `i` field was demoted to a local variable declaration and initialized just as in class level (step 3).

At method level, we identified that some methods return `Iterable` instances. These `Iterable` instances were anonymous inner instances that are defined in placed. These can be replaced for anonymous `Generator` instances being defined in place. The content of the `generate` method can be implemented as previously described. Furthermore, there were other methods returning `Iterable` or `Iterator` but the implementation was too specific to recognize a pattern.

```
1  public class LogicComponentList<T extends AbstractLogicComponent>
2  extends Generator<T> {
3   ...
4
5   @Override
6   public void generate() {
7    int i = 0;
8    while (i < this.getSize()) {
9     yieldReturn(this.get(i));
10    i++;
11   }
12  }
13
14  ...
15 }
```

**Figure 6.16:** Translation of the class depicted in Figure 6.15 into the generator construct.

## 6.5 Case Study

This case study is inspired by a presentation [15] at PyCon'2008, where tricks were presented to use Python generators in systems programming. We borrowed some of the examples and ideas that were illustrated in this presentation and applied them in Java by using our extension of generators in Java. In this case study we assess the performance and applicability of generators.

### 6.5.1 Background

An Apache web server [14] provides facilities to get feedback about the activity and performance of the web server. One of these facilities is the access log. The web server keeps track of all the requests processed in an access log file in the form of records. These records are lines holding information of a request done to the web server (see Figure 6.17). As we can see from this figure, a record contains the IP address of the client that made the request, the identity of the client or a hyphen "-" to indicate that the information was not available, the user id of the person requesting the document, the time that the request was received, the request line from the client, the status code sent back by the server, and the size of the object returned to the client in bytes.

Suppose we would like to compute the total amount of bytes sent back for some monitoring purpose. This would require to parse the access log file, extract each line (record) and finally make the information in each line easily accessible. In order to make this information accessible, it is necessary to parse the line, extract the content and storing it in an accessible data structure. This would allow us to access the number of bytes sent back per record, so we can keep track of the total amount. In our experimental test, this is exactly what we did. But instead of parsing only one file we parsed multiple files.

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

**Figure 6.17:** Example of a log entry in the Common Log Format(CLF).

### 6.5.2 Experimental Setting

For the execution of this experiment, there were two settings. Both settings run three implementations that parse the log files and compute the total amount of bytes. The first implementation does not use the generator construct in its algorithm. We call this implementation the *Common Approach* (see Figure 6.18). The second implementation uses the generator construct, but makes use of the asynchronous (threaded) approach (see Section 5.1.1). Hence we have called this implementation the *Asynchronous Approach*. Finally, the third implementation makes use of the bytecode trasformation approach (see Section 5.2), where the class files are post-processed after normal compilation. We have called this implementation the *ASM Approach*. The implementation for the *Asynchronous* and *ASM Approach* is the same. The difference lies in the support for the generator construct. The implementation consists of three generators: a file generator, a line generator and the byte token value generator. Figures 6.19, 6.20 and 6.21 respectively show their implementation. Additionally, Figure 6.22 shows the usage of the generators in client code to compute the total amount of bytes. As we can see, the client code consists only of 10 lines of code compared to the *Common Approach*. The functionality implemented in the *Common Approach* has been nicely decoupled in generators providing the specific functionality. This has the advantage of separating concerns and making it easier to add additional features where necessary.

Furthermore, each of the implementations in this experiment does the computation for 10 access log files, then for 20 access log files and finally for 100 access log files. Each log file has a size of 637KB and contains 7298 lines (records). The difference between both settings is that one runs the JVM in client mode [3] and the other in server mode [4]. The machine specifications are as follows:

**Os:** Windows XP Professional 64 version 2003 Service Pack 2

**Cpu:** Intel Core 2 CPU 6300 at 1.86GHz

**Hd:** WDC WD800JD-75MSA3 (74.50 GB)

**Mem:** 2.00 GB RAM

**Jdk:** 1.6.0.0_05

[3] java -client. This is the standard modus when running the JVM.
[4] java -server. This option tells the JVM to perform additional optimizations.

```
1  private static long bytesSentToRequest(int numberOfFiles) {
2   long sum = 0;
3   int filesRead = 0;
4
5   File dir = new File("logs");
6   if (!dir.exists() || !dir.isDirectory()) {
7    throw new RuntimeException(dir + "is not a directory or doesn't exist.");
8   }
9
10  for (String fileName: dir.list()) {
11   if (filesRead == numberOfFiles) {
12    break;
13   }
14   filesRead++;
15   File file = new File(dir.getAbsolutePath() + FILE_SEPARATOR + fileName);
16   Scanner scanner = null;
17   try {
18    scanner = new Scanner(file);
19    while (scanner.hasNextLine()) {
20     String line = scanner.nextLine();
21     String[] tokens = line.split(" ");
22     String byteToken = tokens[tokens.length - 1];
23     long byteValue = byteToken.equals("-")? 0 : Long.parseLong(byteToken);
24     sum += byteValue;
25    }
26   } catch (FileNotFoundException e) {
27    e.printStackTrace();
28   } finally {
29    scanner.close();
30   }
31  }
32
33  return sum;
34 }
```

**Figure 6.18:** Excerpt of the algorithm used in the implementation that parses the log files and compute the total amount of bytes sent by the web server.

### 6.5.3 Microbenchmarking

In order to assess the performance of each implementation, careful consideration has been taken. It is well known that microbenchmarking [16], as it is commonly known, can lead to many misleading results if it is not well done.

The difficulty when trying to measure the performance of Java applications comes from the environment in which these applications are run, the JVM. Current JVMs run Java applications with a set of tools that enable them to run these applications faster by performing optimizations. The JIT compiler plays an important role here. JIT compiler techniques are becoming more and more accurate at performing optimizations on the code being run. Code that is being frequently called will be compiled into native machine code by the JIT compiler. The process of identifying frequently executed code and compiling it, adds additional execution time spent by the program being run. Therefore the JVM can be warmed up by running the application through a number of iterations removing the "noise" created by the

```
1  /**
2   * Generates files from a directory.
3   */
4  public class FileGenerator extends Generator<File> {
5   private final String dir;
6   private final int numberOfFiles;
7
8   private static final String FILE_SEPARATOR = System.getProperty("file.separator");
9
10  public FileGenerator(final String dir, final int numberOfFiles) {
11   this.dir = dir;
12   this.numberOfFiles = numberOfFiles;
13  }
14
15  @Override
16  protected void generate() {
17   File directory = new File(dir);
18   int filesRead = 0;
19   if (directory.exists() && directory.isDirectory()) {
20    for (String fileName: directory.list()) {
21     if (filesRead == numberOfFiles) {
22      break;
23     }
24     filesRead++;
25     yieldReturn(new File(directory.getAbsolutePath() + FILE_SEPARATOR + fileName));
26    }
27   } else {
28    throw new RuntimeException(dir + " is not a directory or it doesn't exist.");
29   }
30  }
31 }
```

**Figure 6.19:** Generator used to generate all the access files located in a specific directory.

process of identifying and compiling frequent run code into native machine code. The performance measurements after this warm up session will be more consistent and reveal an execution time of the application closer to reality.

Another issue is the ability of the JIT compiler to perform other optimizations. These optimizations concern dead code removal, method inlining, and many others. It is this reason why writing and interpreting microbenchmarks for dynamically compiled languages is far much more difficult than for statically compiled languages. The JIT compiler is continuously gathering profile information on the code and performing optimizations at unexpected points during the run of the program. Most of the programs that are written for microbenchmarking do actually nothing, which is detected by the JIT compiler in the JVM. This is optimized away, leading to misinterpretations of the code being run. Finally, allocated objects in the program will be garbage collected at some point. The run of the garbage collector can distort timing results, hence the time spent by here must be accounted for.

For this experiment, we used the `System.nanoTime()` call in Java to time the application. Furthermore, each implementation was run with the option `-verbose:gc` to gather

```
1  /**
2   * Generates the lines from an access file.
3   */
4  public class LineGenerator extends Generator<String> {
5   private FileGenerator fileGenerator;
6
7   public LineGenerator(FileGenerator fileGenerator) {
8    this.fileGenerator = fileGenerator;
9   }
10
11  @Override
12  protected void generate() {
13   for (File file: fileGenerator) {
14    Scanner scanner = null;
15    try {
16     scanner = new Scanner(file);
17     while (scanner.hasNextLine()) {
18      yieldReturn(scanner.nextLine());
19     }
20    } catch (FileNotFoundException e) {
21     e.printStackTrace();
22    } finally {
23     scanner.close();
24    }
25   }
26  }
27 }
```

**Figure 6.20:** Generator used to generate the lines (records) from the access files.

information about the time spent by the garbage collector during the application run. This information was gathered and the amount of time spent by the garbage collector was subtracted from the total amount of time measured. This adjustment yields the amount of time spent by the application only. Finally, for each run of a implementation a warm up session was done in order to counter for any compilations and optimizations done by the JVM. To this end, we run each implementation ten times to warm up. This was followed by another run of ten times where we took the average of the measurements. We chose to run the implementation ten times as from manual runnings of the application, we found that with this number of iterations, timing results become stable for this application. This was achieved by collecting data on the run of the program while running the JVM with the `-XX:+PrintCompilation` flag.

### 6.5.4 Results

The results are listed in Figures 6.23, 6.24 for the runnings in client mode and in Figures 6.25, 6.26 for runnings in server mode. All the results were converted from nanoseconds to seconds as this is more convenient for readability. From the results it is clear that the performance is linear in relation to the amount of log files to parse for both client and server mode. Furthermore, the speedup of the ASM Approach over the Asynchronous Approach

```
1  /**
2   * Generates the byte token value from a given entry log.
3   */
4  public class ByteTokenGenerator extends Generator<Long> {
5   private LineGenerator lineGenerator;
6
7   public ByteTokenGenerator(LineGenerator lineGenerator) {
8    this.lineGenerator = lineGenerator;
9   }
10
11  @Override
12  protected void generate() {
13   for (String line: lineGenerator) {
14    String[] tokens = line.split(" ");
15    String byteToken = tokens[tokens.length - 1];
16    long byteValue = byteToken.equals("-")? 0 : Long.parseLong(byteToken);
17    yieldReturn(byteValue);
18   }
19  }
20 }
```

**Figure 6.21:** Generator used to generate the byte token value in an entry log of an access log.

```
1  private static long bytesSentToRequest(int numberOfFiles) {
2   long sum = 0;
3
4   FileGenerator fileGenerator = new FileGenerator("logs", numberOfFiles);
5   LineGenerator lineGenerator = new LineGenerator(fileGenerator);
6   ByteTokenGenerator byteTokenGenerator = new ByteTokenGenerator(lineGenerator);
7
8   for (long byteValue: byteTokenGenerator) {
9    sum += byteValue;
10  }
11
12  return sum;
13 }
```

**Figure 6.22:** Excerpt of the client code that makes use of the generators to compute the total amount of bytes sent back by a web server.

is nearly two. This means that our implementation of the generator construct is twice as fast as the asynchronous approach that makes use of threads. When comparing the speedup of the ASM Approach over the Common Approach we can see that our implementation of the generator construct is slightly faster for client mode and lightly slower in server mode. However the amount by which our implementation is faster or slower is so small that it is negligible. Therefore, we conclude that using the generator construct is at least as fast as a similar implementation without using the construct. As a final remark, it can be said from the results that running the JVM in server mode is at least 20% faster.

| Log Files | CA | AA | ASM A | Speedup over CA | Speedup over AA |
|---|---|---|---|---|---|
| 10 | 0,0853 | 0,1812 | 0,0818 | 1,043 | 2,216 |
| 20 | 0,1687 | 0,3593 | 0,1621 | 1,041 | 2,217 |
| 100 | 0,8200 | 1,7690 | 0,7849 | 1,045 | 2,254 |

**Figure 6.23:** Performance measurements for three implementations that count the total number of bytes sent to requests for a given number of access log files. **CA**=*Common Approach*, **AA**=*Asynchronous Approach*, **ASM A**=*ASM Approach*. All measurements were collected in nanoseconds but converted to seconds for readability. These results correspond to the running of the JVM in client mode.

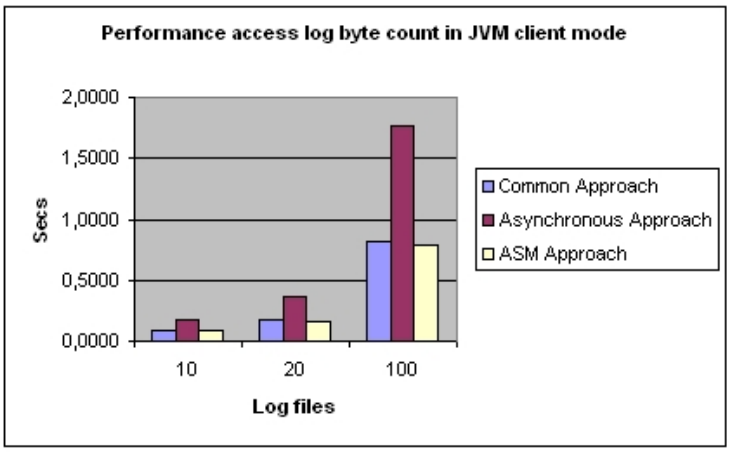


**Figure 6.24:** Performance of the access log byte count in the JVM client mode.

## 6.6 Comparison with the Informancers Collection Library Implementation

As part of the evaluation, we have compared our implemented solution with the implementation of the Informancers Collection Library (see Section 4.2.2). The implementation is similar in the strategy and technology used.

Both implementations require to extend an abstract class to define generators: `Yielder` for the Informancers implementation and `Generator` for our implementation. Furthermore, both classes require to override a method: `yieldNextCore` for the Informancers implementation and `generate` for our implementation. In each method a call to `yieldReturn` must be done to return values. Additionally, the Informancers implementation provides a `yieldBreak` call to interrupt the execution of the `yieldNextCore` method. Recall that this implementation is based on the ideas of Python and C#. C# does not allow `return` statements in generators. However, this is not necessary for the implementation of generators by means of the abstract class in the Java programming language. Hence, we have not intro-

| Log Files | CA | AA | ASM A | Speedup over CA | Speedup over AA |
|---|---|---|---|---|---|
| 10 | 0,0642 | 0,1406 | 0,0651 | 0,986 | 2,158 |
| 20 | 0,1208 | 0,2634 | 0,1331 | 0,907 | 1,979 |
| 100 | 0,5848 | 1,2624 | 0,5835 | 1,002 | 2,163 |

**Figure 6.25:** Performance measurements for three implementations that count the total number of bytes sent to a request for a given number of access log files. **CA**=*Common Approach*, **AA**=*Asynchronous Approach*, **ASM A**=*ASM Approach*. All measurements were collected as nanoseconds but converted to seconds for readability. These results correspond to the running of the JVM in server mode.

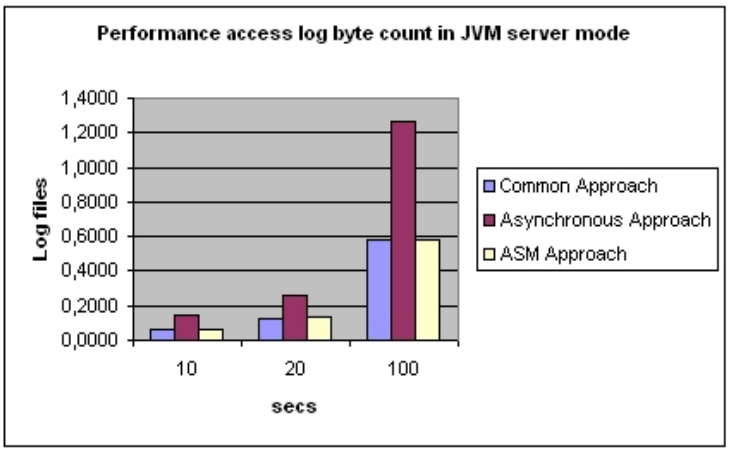


**Figure 6.26:** Performance of the access log byte count in the JVM server mode.

duced a `yieldBreak` call as this is just equivalent to a `return` statement which is allowed in the `generate` method.

The Informancers Collection Library makes use of the bytecode weaving approach just as our implementation. The difference lies in the point at which the transformations are done. Our solution perform the transformations as a post-compilation step by running the post-processing tool on a given class file while the Informancers implementation perform the transformations just before the class is going to be loaded by the JVM at runtime. Hence, our post-processing tool adds compilation overhead while Informancers implementation does this during runtime.

The technology used by the Informancers implementation to perform the transformations on the bytecode is the same. The ASM framework has been employed to implement the transformations. The strategy applied to support generators by the Informancers implementation is as follows:

1. Introduce a `state` field to record the state of the control flow in the `yieldNextCore`

method.

2. Insert a lookup table at the beginning of the `generate` method that transfers control to the right point upon resumption of the `yieldNextCore` method.
3. Remove all the local variables from the `yieldNextCore` method.
4. Replace each reference to a local variable by a reference to its field variant.

As can be noted this strategy is similar to that one used by our implementation. Our implementation uses a `state` field in combination with a lookup table for handling the control flow within the `generate` method. In the Informancers implementation all local variables are removed from the `yieldNextCore` method and their usage is replaced by fields that are introduced at class level. Furthermore, all introduced fields are non-primitive types. This means that the replacement of references to primitive typed local variables require extra instructions in bytecode for boxing and unboxing with additional `CHECKCAST` instructions. We found that references to fields that replace references to non-primitive typed local variables (objects) which are used for method calls require an extra `CHECKCAST` instruction. This approach leads to performance degradation as opposed to our implementation where we record and restore the state of local variables where necessary. In our approach, we introduce fields with primitive types for those variables with primitive types. Hence no additional instructions are needed for boxing or unboxing. Furthermore, `CHECKCAST` instructions are only introduced where they are actually needed. We make in our implementation a distinction between `generate` methods with `try\catch\finally` blocks where we replace all references to local variables by their field variants. For those `generate` methods that do not contain `try\catch\finally` blocks, local variables are used where only their state preservation is handled by storing and restoring values to and from their fields variants.

The tests described in Section 6.1 were run with the implementation of the Informancers Collection Library. The result was that this implementation does not handle `try/catch/finally` blocks properly as none of the tests for `try/catch/finally` blocks run successfully. We also found out that the lifting of arrays that were initialized with the curly braces syntax was not done properly as the tests for this case did not pass.

Finally, debugging support was not properly done by the Informancers implementation. The execution of a program making use of generators could be followed in a debugger. Hence the line numbers were not an issue. Recall that one of the steps of the applied strategy was to remove local variables from the `yieldNextCore` method. Another one was to replace all references to local variables by their field variants. This has as result that neither the local variables nor their values can be traced back while debugging an application that makes use of generators. The values can be traced back by exploring the fields of the generator instance, but this is not transparent to the user as the names of the fields are cryptic (i.e. `slot$1`) and viewing the values needs to be done via a deviation. This as opposed to our implementation where debugging support if fully supported for generators.

# Chapter 7

# Discussion

In this thesis work we set out to extend the Java programming language with the generator construct. To this end, we started with a clear outline of the problem statement and its requirements. This was followed by a thorough discussion of the implemented solution along with its evaluation. In this chapter, we reflect on the achieved results and their compliance with the requirements for this thesis project and provide a discussion with a broader view on the implemented solution by generalizing the employed approach.

## 7.1 Reflection

During this thesis we strived to provide a solution that would be as non-intrusive as possible within the constraints posed by the requirements in this thesis project. One of the most important aspects of the implemented solution is that it does not break any existing code and offers developers enough freedom in their choice for the tooling used in the development of applications.

We managed to implement a solution that complies with all the requirements. The implemented solution does not depend on any specific IDE (R1). The extension was based in terms of the existing language's syntax (R2). Any compiler of choice can be used (R3). No specific flags are required for running the JVM with the resulting bytecode (R4). The solution is transparent to the developer (R5) as it can be used as a Java native feature. The semantics for the generator construct have been well defined in this document (R6) and were thoroughly tested. Debugging support for the generator construct has been fully implemented (R7).

There was a concern of performance degradation by using generators. Hence, it was a requirement that the usage of generators should have acceptable performance (R8). As we could see from the performance results of the case study, using the asynchronous approach causes a performance degradation of factor two in comparison to an application that provides the same functionality but that does not make use of generators in its implementation. For the approach that make transformations in the bytecode the performance was

almost the same. The performance difference was almost negligible. Hence, we conclude that we comply with the performance requirement as the bytecode transformation approach provides acceptable performance. The choice is left to the developer to make a trade off between altering the building process or accepting the performance degradation but leaving the build process intact.

A comparison between the implemented solution and the implementation of the Informancers collection library was drawn. Both implementations are similar in terms of concepts and technology used. However, the Informancers collection library implementation is intrusive in that it requires the user to run the JVM with special flags (R4) and does not provide debugging support (R7). Furthermore, not all possible cases are handled correctly as from the test results it appears that `try\catch\finally` blocks do not work correctly in generators nor the lifting of array variables that are initialized with the curly braces syntax was done properly. Finally, the strategy applied to perform the transformations leads to bytecode that might have performance issues (R8) due to implementation of this steps.

To conclude this section, it can be said that the implemented solution complies with all the requirements as has been described in this section. The implemented solution is planned to be used by TOPdesk's development team.

## 7.2 Generalization of the Employed Approach

In the introduction section of this document, we defined the problem statement where we posed two research questions to guide the design and implementation of a solution based on the requirements. In this section we try to provide an answer for this questions.

The first question was formulated as follows:

**Q1:** How to extend the Java programming language with a generator construct non-intrusively?

In this document we defined non-intrusive as not posing any restrictions to the tooling being used in the standard language and the environment in which it is used. This was more clearly formulated further in the document as a set of requirements in Section 4.1. The answer to this question is the result of the implemented solution during this thesis work.

**Q2:** How to develop a language extension so it can be successfully introduced in a non-intrusive manner?

In order to answer this question we need to define the term non-intrusive more specifically. The development of a new extension to the Java programming outside the formal

channels[1] means that good care needs to be taken not to break any existent Java source code and make sure that the solution is compatible with any future enhancements that are made to the language through the previously mentioned channels and that all the conventional tools used to develop the existent language can be used with with the extension as well. Hence, these constraints that define the term non-intrusive can be more precisely formulated as the following requirements which we will use to discuss a general solution:

**R1:** The implemented solution should be IDE-independent.

**R2:** The new introduced extension should be based on the already existing language's syntax.

**R3:** A solution should be independent of any particular Java compiler.

**R4:** The usage of the extension should not require to run the JVM with any specific flags to provide its support.

**R5:** The solution should be transparent to the developer.

**R6:** The semantics of the new extension should be well defined.

**R7:** The new extension should provide debugging support.

**R8:** The performance of an application making use of the extension should be acceptable conform to the aim of the extension.

As we can see, these are the same requirements as those for the solution of extending the Java programming language with the generator construct but they have been generalized so they apply to any extension that needs to be implemented. Furthermore, they offer the same advantages as described in Section 4.1.

After having formally defined the term non-intrusive in the form of requirements, we proposed the solution depicted in Figure 7.1. This solution is based on the ideas and experiences gained while implementing the extension of the generator construct into the Java programming language. In the figure we have a post-processing tool. This post-processing tool must be integrated into the build process as a post compilation step. The post-processing tool requires only class files containing bytecode on which transformations are performed in order to support the desired extensions into the Java programming language. The produced result of the post-processing tool is bytecode that supports the intended extensions.

The internal architecture of the post-processing tool as depicted in Figure 7.1 allows flexibility and separation of concerns for the implementation of an arbitrary number of extensions. The extension handler recognizes the usage of new extensions and delegate the

[1] The Java Community Process (JCP), http://jcp.org

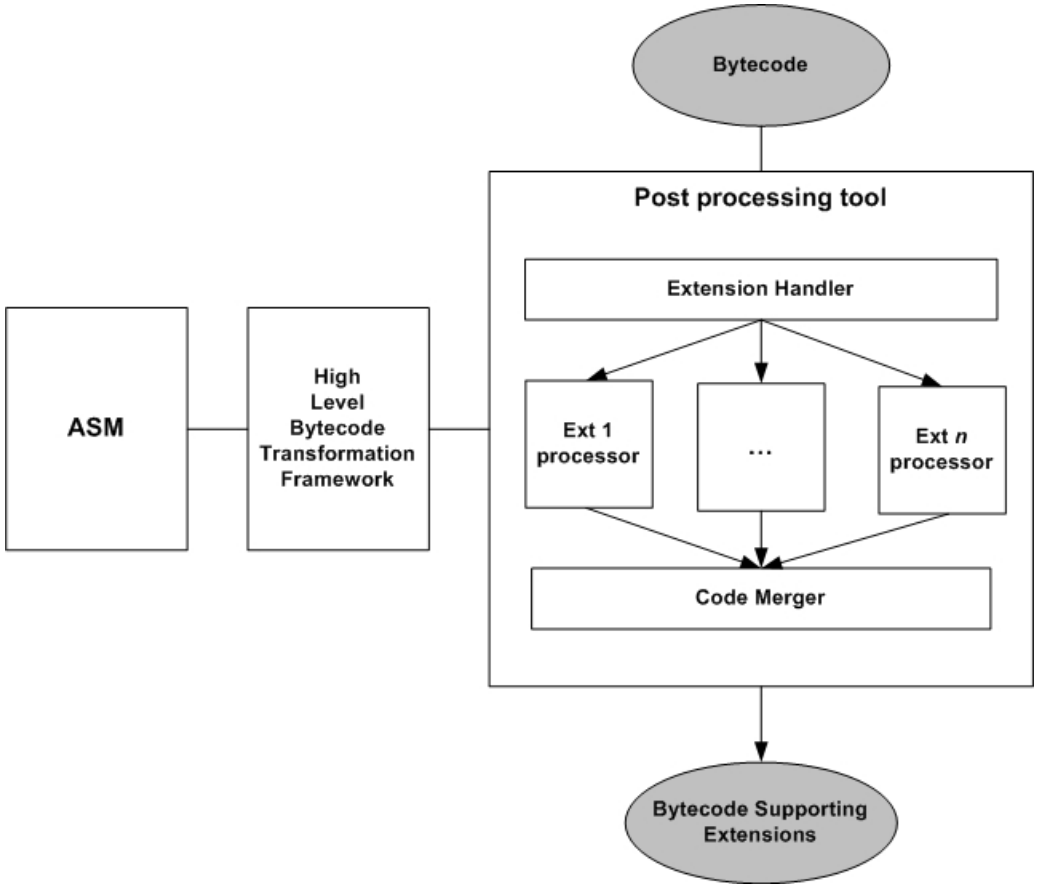


**Figure 7.1:** Architecture of the generalized proposed solution.

transformations that need to be performed to the specific extension processors. Each extension processor has the task of performing the transformations on the bytecode to support the particular extension. By delegating these tasks to the specific extension processor the transformation strategies for each supported extension can be encapsulated. Hence, the implementation of extensions can be done separately. Furthermore, the extension handler should provide an easy way to add new extensions. A possible way to do this would be by implementing a plugin mechanism.

The technology used to perform the bytecode transformations in our proposed solution is the ASM framework. Although other frameworks could be used as well. The key is to have a framework on top of the bytecode manipulation framework that provides a higher level of abstraction to implement the transformations on the bytecode. It was our experience while using the ASM framework that it provides the right tools to perform bytecode manipulations, but the level of abstraction over the bytecodes was too low which made it difficult to perform certain transformations. By having a framework with a higher level of

abstraction on top of the bytecode manipulation framework, the extension processors could benefit from its use since it would be much easier to implement the required transformations. Finally, the code merger is in charge of performing any necessary steps to complete the transformation of the bytecode which are not specific to any extension.

By developing a tool with the described architecture, we can comply with all the requirements and hence provide a non-intrusive solution for any extension in mind. Since the tool is to be used as a post-compile step that needs to be integrated in the build process, the solution can be transparent (R5) to the user. It can be used in combination with any compiler (R3) and the JVM can be run with no special flags (R4). Because of this, the solution can be integrated into any IDE (R1), provided that the syntax of the new extension is based in terms of the existing Java language's syntax (R2). The specification of the semantics for the new extension (R6) is to be done by the implementor of the extension processor as well as the support for debugging (R7) and acceptable performance (R8).

# Chapter 8

# Conclusion and Future Work

In this thesis work, we have designed, implemented and evaluated a solution for extending the Java programming language with generators. The solution is based around the concept of non-intrusiveness. This concept has been properly formulated as a set of requirements. These requirements specify that the usage of the extension should still allow developers to choose the tooling and environment for the development of applications in the Java programming language. The solution is implemented as a hybrid approach. This approach consists of an asynchronous approach where the support for generators is based on threads and on another that relies on bytecode transformations. Both approaches provide support for generators independently from each other. The choice is left to the user who needs to make a trade off between performance degradation without altering the build process and satisfactory performance with an alteration in the build process. The implementation has been thoroughly tested in terms of expected behaviour by using generators in different cases. Furthermore, we have shown the applicability of generators and assessed their performance.

We have discussed other existing solutions and provided a motivation in terms of the requirements why these solutions are not suitable. We have thoroughly compared our solution with the Informancers collection library implementation as it is similar to our solution. The results are that the Informancers implementation do not work properly in all cases, has performance issues and does not provide full debugging support. Our implemented solution complies with all the requirements as discussed in Section 7.1. It is for this reason that we conclude that our implemented solution is non-intrusive and therefore a preferable choice to employ the use of generators.

Future work that follows from this thesis is the research of applications of generators. Other future work is the implementation of the proposed generalized solution for integrating extensions into the Java programming language in a non-intrusive manner. In the proposed solution, an architecture is discussed that facilitates the implementation of extensions by means of extension processors. Further research is needed to investigate how a solution can be developed that allows extensions to be added in a pluggable manner. Hence, the scalability of the proposed generalized solution needs to be investigated. We also mentioned that the implementation of extension processors should benefit from a framework on top of

the bytecode manipulation framework. There is a need to develop a framework that provides a higher level of abstraction to implement bytecode transformations with less effort. Therefore, research needs to be done in order to design and implement a solution for such a framework. Finally, as the generalized solution is based on defining extensions in terms of the existing language syntax, we need to investigate the limitations of this approach.

# Bibliography


[1] Chaotic Java, blog article: How to write Iterators REALLY fast, 2008. http://chaoticjava.com/posts/how-to-write-iterators-really-really-fast/.

[2] The Da Vinci Machine Project. A Multi-language Renaissance for the Java$^{TM}$ Virtual Machine Architecture, 2008. http://openjdk.java.net/projects/mlvm/.

[3] Informancers Collection Library home page, 2008. http://code.google.com/p/infomancers-collections/.

[4] The Java$^{TM}$ Tutorials. About the Java Technology, 2008. http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html.

[5] The JRuby Project Homepage, 2008. http://jruby.codehaus.org/.

[6] The Jython Project Page, 2008. http://www.jython.org/Project/.

[7] Martin Fowler on Internal DSL Style. From martinfowler.com, 2008. http://martinfowler.com/bliki/InternalDslStyle.html.

[8] The Python Programming Language, Official Website, 2008. http://www.python.org/.

[9] The Ruby Programming Language Home Page, 2008. http://www.ruby-lang.org/en/.

[10] Stack Machines discussion, Wikipedia, 2008. http://en.wikipedia.org/wiki/Stack_machine.

[11] Sun Microsystems Inc. Java SE Hotspot Virtual Machine Home Page, 2008. http://java.sun.com/javase/technologies/hotspot/.

[12] Sun Microsystems Inc. Official Home Page, 2008. http://www.sun.com/.

[13] Threaded Implementation of Generators. Adrian Kuhn Blog post, Yield 4 Java, 2008. http://smallwiki.unibe.ch/adriankuhn/yield4java.

[14] Apache http server project home page, 2009. http://httpd.apache.org/.

[15] Generators Tricks for Systems Programmers, 2009. http://www.dabeaz.com/generators/.

[16] IBM article on java microbenchmarking, 2009. http://www.ibm.com/developerworks/java/library/j-jtp02225.html.

[17] JUnit home page, 2009. http://www.junit.org/.

[18] Topdesk B.V. home page, 2009. http://www.todesk.com/.

[19] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[20] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383. ACM, 2004.

[21] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems, 2002. Adaptable and Extensible Component Systems, http://asm.objectweb.org/.

[22] Eric Bruneton. ASM 3.0, A Java bytecode engineering library. ASM 3.0 API documentation, http://download.forge.objectweb.org/asm/asm-guide.pdf.

[23] Markus Dahm. Byte Code Engineering. In *Proceedings JIT'99*, pages 267–277. Springer-Verlag, 1999.

[24] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The Case for Virtual Register Machines. In *IVME '03: Proceedings of The 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49. ACM, 2003.

[25] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18. ACM, 2007.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java$^{TM}$ Language Specification, The (3rd Edition) (Java Series)*. Addison Wesley, 2005.

[28] Guy L. Steele Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[29] Lennart C. L. Kats, Martin Bravenboer, and Eelco Visser. Mixing Source and Bytecode. A Case for Compilation by Normalization. In G. Kiczales, editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, Nashville, Tenessee, USA, October 2008. ACM Press.

[30] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of Aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[31] Tim Lindholm and Frank Yellin. *The Java$^{TM}$ Virtual Machine Specification, The (2nd Edition) (Java Series)*. Prentice Hall PTR, 1999.

[32] Barbara Liskov. A History of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147. ACM, 1993.

[33] Jed Liu and Andrew C. Myers. JMatch: Java Plus Pattern Matching. Technical Report TR2002-1878, Computer Science Department, Cornell University, October 2002. `http://www.cs.cornell.edu/projects/jmatch`.

[34] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127. Springer-Verlag, 2003.

[35] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin/Heidelberg, 2003.

[36] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[37] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159. ACM, 1997.

[38] Nicolas Petitprez Renaud Pawlak, Carlos Noguera. Spoon: Program Analysis and Transformation in Java. Technical Report 5901, INRIA, 2006.

[39] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 1st edition, 2000.

[40] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A Class-Based Macro System for Java. In *Reflection and Software Engineering*, pages 117–133. Springer Berling/Heidelberg, 1999.

[41] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: an Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[42] Bill Venners. *Inside The Java 2 Virtual Machine*. 2nd Revised edition.

# Appendix A

# Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

**API:** Application Programming Interface

**AST:** Abstract Syntax Tree

**DSL:** Domain-Specific Language

**IDE:** Integrated Development Environment

**JIT:** Just-In-Time-Compiler

**JRE:** Java Runtime Environment

**JVM:** Java Virtual Machine

**LOC:** Lines Of Code

**OOP:** Object-Oriented Programming