ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 2, No. 3, 1711-1726 (2017)
www.astesj.com
Special Issue on Recent Advances in Engineering Systems

ASTES Journal
ISSN: 2415-6698

Pluggable Controllers and Nano-Patterns in Java with Lola

Yossi Gil [1], Ori Marcovitch[1], Matteo Orrú[*,1], Ori Roth[1]
[1]*Computer Science Dept., CS Taub Building, The Technion—IIT, Haifa 3200003, Israel*

ARTICLEINFO
Article history:
Received: 11 June, 2017
Accepted: 21 July, 2017
Online: 09 September, 2017
Keywords:
Programming Languages
Preprocessors
Macro Languages
Pluggable Controllers
Language Augmentation
Nano-Patterns

ABSTRACT

Pluggable controllers are a different way to design control constructors such as `if`, `while`, `do`, *switch, and operators such as short circuit conjunction (*`&&`*) and the "*`?.`*" operator of the Swift programming language. Adoption of pluggable controllers enables the final user to modify and extend the control flow constructs (*`if`, `while`*, etc.) of an underlying programming language, the same way they can do if they implement functions such as* `printf` *and class* `String` *in a standard library.*

In modular, pluggable controller based language design, beside core control constructors, there are others, defined in standard libraries, with the purpose of augmenting and enriching the language. These pluggable controllers are extensible and replaceable. Being less intertwined in the main language, control constructor libraries can evolve independently from it, and their releases do not mandate new language releases.

We illustrate the implementation of pluggable controllers using Lola*, a powerful language-independent preprocessor and macro language. We demonstrate the introduction of new pluggable controllers with two case studies. The implementation of a Java stenography based on prevalent Java idioms, called "nano-patterns" or nanos, and the introduction in Java of new code constructs inspired by the Mathematica language's commands.*

1 Introduction

A recent publication [1] described a preliminary new perspective of viewing keywords such as `if` and `while` as library functions such as `printf`: standardized, but user extendable and replaceable. In this work, we expand on this vision, demonstrating the extension of Java with control keywords for two applications: a stenography for a concrete nano-patterns language whose prevalence in Java was previously demonstrated [2] and the implementation of Mathematica-like control structure in Java.

1.1 Control Constructors

Control Constructors are defined as those elements of a programming language that make it possible to assemble *commands*. In textbooks [3–5] as well as in classic works on programming languages [6–9], is reported that there are essentially three kinds of control constructors:

| ***Sequential*** | ***Iterative*** | ***Conditional*** |

A further distinction can be made between *atomic* and *compound* commands, where the latter are formed by primitive and other smaller compound commands. In Pascal, for example, atomic commands are the empty command, the assignment, and the procedure call. On the other hand, `Begin`...`end` is the sequential constructor, whereas `While`...`do`... is an iterative constructor, and, `If`...`then`... is a conditional constructor.

In real languages, there is no sharp separation between *expressions* and *commands*. Many *operators* that form expressions can act as control constructors. This is the case of standard short circuits operators such as "`&&`" and "`||`", and standard conditional operators such as "`· ? · : ·`", Other examples worth mentioning are represented by:

1. The "`,`" (comma) operator of C (the sequential comma operator of C).
2. The "`||`" operator, which "provides a default value", the Python variant of a short circuit.
3. The "`??`" operator (`null` coalescing) of C#.
4. The "`?.`" operator (*fluency on* `null`) of Swift.
5. The "`noexcept`" operator of C++, recently introduced, which guards against exceptions.

[*]Computer Science Dept. The Technion—IIT, Taub Building, Haifa 3200003, matteo.orru@cs.technion.ac.il

These operators can be expressed in terms of sequential, iterative, and conditional control constructors (henceforth, "*SIC*"). *Structured programming* [7] is defined by the notion of SIC, meaning that SIC are sufficient to impose structure on the unstructured. In fact, any program that has **goto** instructions in it can automatically be converted into an equivalent one that uses only SIC [10].

Concrete languages sometimes deviate from the traditional concept of SIC. A typical example is the **while** command of Python that has its peculiar **else**. Another example is represented by the different ways **switch** statements deal with fall-through cases, which is different in some languages. Variations and diversifications also appear in the semantics of

try...**catch**...**finally**

blocks. Language engineers, developers and practitioners are generally interested in using new language features that are likely to enhance their productivity, efficiency and reduce their errors. In general, the creativity of language engineers slows down after a languages' first release. Even small changes to a (successful) language definition might have unpredictable, potentially negative effects in the least expected places [11, p.497–508].

For the above reason, the long time taken before introducing the **switch**ing on strings feature in Java (see Fig. 1) is understandable. It took around twenty years from proposal[1] to implementation[2]—probably because of worries about its implications for engineers.

```
public static void main(String[] args) {
  for (String arg: args)
  if (arg.equals("-c") || arg.equals("--bytes"))
  . . .
  else if (arg.equals("-m") || arg.equals("--chars"))
  . . .
  else if (arg.equals("-w") || arg.equals("--words"))
  . . .
  else if (arg.equals("-l") || arg.equals("--lines"))
  . . .
  else if (arg.equals("--help"))
  . . .
  else if (arg.equals("--version"))
  . . .
  . . .
}
```

Figure 1: Java code with multiple comparisons of the same string variable with a sequence of string literals.

The drawback of the prudence characteristic of language architects, when it comes to introducing new features, is that it might negatively affect software systems. For example, consider that the late introduction of generics in Java forced developers to use the unsafe **Vector** as a substitute.

1.2 A Modular, Plugin-Oriented Approach

This paper raises the idea that the design of programming languages, specifically in respect to their control constructors' design, may follow a different, modular *plug-in* oriented approach. This proposal suggests that a programming language should have a number of core control constructors, whereas additional and more sophisticated control constructors, which we called *pluggable controllers*, can be defined in standard libraries of controllers, similar to what happens with functions such as **printf** in C or classes such as **String** in Java.

This approach promotes the decoupling between the controllers and the language architecture, leaving to both sides the freedom to evolve independently. This solution has advantages in different scenarios. The final user would be able to modify and extend the control flow constructs such as **if**, **while**, etc. of an underlying programming language. Language designers would be able to experiment new features before changing the language architecture. The proposed approach could also foster more sound discussions on the introduction of new features inside the users community.

The problem of the modularization of languages components is he main problem of *Modular Language Development*, a research branch that investigates how to componentize language design. Several solutions to the problem of componetization are available—see, for example, Cazzola and Vacchi [12] which adopt a solution based on *Traits* [13].

Another approach exploits the concept of *extensible language*, which presents some challenges such as that of adapting the parser according to language evolution [14] .

1.3 Lola

The imposition of a new code constructors on an underlying language can be done using Lola [15], the ***L**anguage **O**f **L**anguage **A**mendment*, which is a modern, language-independent, preprocessor and macro language, orientated to *language extension*. Lola makes it possible to augment and amend syntactical constructs of any host language.

Lola works like a filter [16] (see also [17, Sect. 3.1]) that converts an input stream comprising tokens of different kinds into an output stream. Lola allows designers to extend and enrich existing languages with new constructors without affecting languages architecture, since plugged controllers can be defined in Lola configuration files. In addition, with Lola, experiments on new language features can be conducted in controlled environments by scholars, designers and practitioners.

1.4 Nano-Patterns

A *nano-pattern* is a recurring solution adopted by developers, for a common task that involves control constructors. The fact that they are recurring solutions means that they are used often by developers. Another possible interpretation is that they are workarounds that developers found to overcome languages restrictions. In a previous study, it has been shown that there exist a number of prevalent nano-patterns. With the term "prevalent" we mean that they recur often in software systems. These solutions are potential candidates for inclusion as new language features—which can be done painlessly with a pluggable controllers approach.

[1] http://bugs.java.com/bugdatabase/view_bug.do?bug_id=1223179

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html

1.5 Contribution

The present paper makes several contributions:

- We propose a novel, modular approach to the design of control constructors, the use of *pluggable controllers*. This approach makes it easier to add new (experimental) features to an existing programming language.
- We introduce Lola, a new preprocessor and macro language, oriented to language augmentation. We illustrate its syntax, work flow and present some meaningful examples.
- We further discuss the concept of *nano-patterns* (from now on *nanos*), introduced in previous works [1] as recurring solutions adopted by developers, to common tasks that involve control constructors.
- We demonstrate how to impose pluggable controllers on top of Java using Lola. Specifically, we present two case studies that show a specific usage of control constructors: a new *stenograpy* for Java nano-patterns and the Java implementation of common Mathematica commands.

Outline: The paper is organized as follows. Sect. 2 illustrates some basic concepts to establish a common vocabulary. Sect. 3 presents Lola, the Language Of Language Amendment. Sect. 4 describes the two case studies above mentioned. Sect. 5 discusses some practical applications and outlines some promising avenues for researchers and practitioners. Sect. 6 reports on related work. Sect. 7 concludes and suggests future directions for this research.

2 Background

2.1 Pluggable Controllers

The following is an intuitive definition of pluggable controllers:

> **Pluggable Controllers**
>
> Controllers should be just like functions and classes found in a library: *standardized*, yet *extendable* and *replaceable*.

The term *controller* in this definition encompasses both classical control constructs such as `while` and `if` ⋯ `else`, *and* operators such as "`??`" that control the order of evaluation of their arguments. According to the pluggable controllers approach, a language has only essential, *built-in* SIC. A standard library of varieties of controllers complements the basic SIC: For example, built-in might be the following:

$$\{c_1; \cdots c_n\} \tag{2.1}$$

or

$$\texttt{while (}e\texttt{)}\ c \tag{2.2}$$

and

$$\texttt{if (}e\texttt{)}\ c_1\ \texttt{else}\ c_2, \tag{2.3}$$

while a standard library contains `for`, `do` and `switch`. In any case, the library is not sealed. On the contrary, developers may add their own control constructors, e.g., adding

$$\texttt{while (}e\texttt{)}\ c_1\ \texttt{else}\ c_2 \tag{2.4}$$

to Java. With pluggable controllers, extending `switch` to strings becomes possible using by library evolution rather than a new language version. Further, no disturbance to the language's core should occur by adding a "`?.`" operator to it. Pluggable controllers join the trend of parameterization of programming languages' elements.

Historically, standard procedures such as `Write` were hardwired in many programming languages, and the same happens today, even in contemporary languages such as AWK. Pascal was the first language to introduce pre-defined procedures such as `Writeln`, functions such as `Sin`, literals such as `true`, and types such as `Integer`. These predefined types, functions, literals or procedures can be overridden by developers when necessary. Likewise, in C, anyone can produce their own version of `printf` from the standard `<stdio.h>` library. In Java, the atomic types, e.g., `int`, `double` and `boolean`, are built-in, whereas their Go equivalent are *pre-defined*.

2.2 Nano-Patterns

The definition of nano-patterns (or simply nanos) was introduced at the end of the seminal work on Micro-Patterns [18] and lately used by authors such as Singer et al. [19] and Batarseh [20], who provided their definitions. In this paper we use the following definition:

> **Nano-Patterns** (*intuitive definition*)
>
> A *nano-pattern* (or *nano*) is a pattern of (typically less than a dozen) control constructs, which recurs frequently, serving common or similar purpose, yet cannot be abstracted easily by a function.

Our working hypothesis was that ordinary control constructors are designed with a prudent view of their usability. At the same time, nanos leverage a unique way of using control constructors that emerges from their function, heretofore not necessarily discerned by language designers. Moreover, the different ways in which control constructors may be used are not static but evolve over time. They are continually being developed and perfected in many specific domains.

A case in point regards the nanos of iterations that were not included in the design of traditional languages. Consider the design of a pluggable controller that captures the following nano[3] in Java code:

$$\texttt{while (}e\texttt{)}\ c_1\ \texttt{else}\ c_2 \tag{2.5}$$

[3] This is actually a control constructor in Python.

Introducing this pluggable controller in Java might be worth the effort depending on the frequency of the patterns using `if` and the auxiliary variables required to capture its behavior. The same applies to other constructs that capture recurring tasks such as "apply an and/or/sum" and other associative operations on `Iterable`s" or "iterate, zipper style, on two lists" A high frequency of reuse may suggest that developers would gain some benefits by the introduction of a pluggable controller. In order to avoid dangerous side effects on the language architecture, the proposed solution is to place controllers in a library rather than in the language core.

In a previous work, a catalog of 38 nanos was identified through a process that includes an initial *subjective* scrutiny and successive further evaluations using the *objective* prevalence threshold test against a *baseline* corpus [2]. All the found nanos are:

- *Traceable*, using an appropriate parser (our nano tracer).
- *Purposeful*, because they are aimed at performing a specific programming task.
- *Prevalent*, according to the definition that follows.

The *Prevalence* of a nano in a given project of a corpus is equivalent to the number of times that it recurs in that specific project. A nano is "prevalent" if its *prevalence* value is higher than a given *prevalence threshold*. The *prevalence threshold* ρ is defined as the fraction of projects of a corpus, for which nano prevalence is higher than a given *reuse threshold*. Here we are considering $\rho = 0.5$, as in reference [2]. The *reuse threshold* r_{th} of a project p is computed similarly to the way done for the popular h-index [21] of a researcher: A project p has a method reuse index r if it has r-methods that are reused r times or more. Thus a nano meets the prevalence threshold criterion if its prevalence is higher than the *reuse threshold* in, at least, 50% of the projects in a corpus C.

The prevalence threshold criterion is robust, objective and filters out irrelevant or domain specific candidate nanos. The reuse index is more robust than other statistics such as *mean* and *median* [22]. In fact, reuse is characterized by a long tail distribution as it happens in many software engineering contexts [23–26]. Being based on the h-index, also the r-index presents the same advantages, apart from the mentioned robustness, such as the fact that it captures the "core" of projects' methods, namely those that are more reused. The insensitiveness towards small values (low reuse of methods) allows to filter out those values that are related to project idiosyncratic features [27]. At the same time, the r-index "inherits" from the h-index some of its drawbacks. When it is used to compare different projects, it is needed to take into account that there are intrinsic differences between projects, such as their age. The *reuse threshold* tends to mitigate some of these drawbacks. Moreover, when comparing the r-indexes of two projects, the only thing that matters is that it refers to the "core" methods. Two projects with the same r-index are treated as equivalent, even if there are marked differences in terms of total number of reuse or maximum number of reuse. From a different perspective, this can be considered an advantage because it means that the r-index implicitly ignores outliers.

Table 1 lists 19 out of the 38 nanos in the original catalog [2]. These nanos are those that involve **SIC**s. This new catalog has two categories: *"Expressions and Command Elaborators"* and *"Operation and Operators on Multitudes"*. Nanos included in the first category are used to manage small errors and exceptional values, substituting missing values with default ones, guarding against null parameters or missing pre-conditions in method execution and handling unusual control flow. The second category covers simple operations on multitudes (i.e., arrays, lists, sets, etc.) including but not limited to: retrieving subsets of values, applying commands, computing cardinality of a multitude, etc. The second column provides the names of the nanos, whereas the third column presents their *Intent*, which is a description of their purpose, written in pseudo code. It is worth noting that often a nanos' name is obtained by juxtaposing the corresponding pseudo code keywords. For example, the following code describes the intent of the nano evaluateUnlessDefaultsTo:

$$[\ \textbf{Evaluate}\ ]\ e\ \textbf{unless}\ b\ [\ \textbf{default}\ e'\ ], \tag{2.6}$$

which captures the following Java snippet of code like:

```
pos > 0 ? pos - 1 : 0
```
(2.7)

The pseudo code adopts the following conventions: C–**c**ommand, e–**e**xpression, p–**p**redicate, M–**M**ultitude (i.e., an array, stream, list, set, collection, etc.), b–**B**oolean expression, i–**i**dentifier, T–**T**ype, and, X–e**X**ception type. The fourth column reports the prevalence. Its values are related to the first, *baseline*, corpus, called Gil-Lalouche Corpus (from now on GL Corpus). It is composed of 26 popular projects selected from the *GitHub's TrendingWikiBookJavaIdiom repositories* [4] augmented by the *GitHub Java Corpus* [5] list due to Allamanis and Sutton [28]. See Table 2 for a list of the projects. This corpus was previously used in other independent research [29].

[4] https://github.com/trending?l=javasince=monthly

[5] http://groups.inf.ed.ac.uk/cup/javaGithub/

Table 1: Language of Java Nano-Patterns

	Name	Intent[a]	Prevalence[b]
	Expression and Command Elaborators		
1	executeUnless	**[Execute]** C **unless** b **[Execute]** C **when** b	100%
2	whenHoldsOn	**When** $p(\cdot)$ **of** x,y,z: C_1 **of** w: C_2 **otherwise** C_3	100%
3	defaultsTo	e_1 **default** e_2 e_1 `??` e_2	81%
4	questionQuestion	e_1 **default** e_2 **else** e_3 e_1 `??` $e_2 : e_3$	96%
5	safeNavigation	e`?.`f e`?.`$m()$	50%
6	evaluateUnless-DefaultsTo	v = **[Evaluate]** e **unless** b **[default** e'**]**	92%
7	safeCast	T`?(`e`)`	96%
8	throwOnFalse	**Throw** X **when** b b `?!` X	100%
9	throwOnNull	**Throw** X **when** e **nulls** e `!!` X	81%
10	notNullRequired	**Return default when** a **nulls** T f`(safe` F_1`,...,safe` F_n`){...};`	88%
11	notNullAssumed	**Return default when** e **nulls** $e \neq$ `null || return default;`	92%
12	holdsOrReturn	**Return default when** b b `|| return default;`	100%
13	ignoringExceptions	{...} `ignoring` $X_1,\ldots,X_n$`;`	58%
14	letInNext	`let` $v \leftarrow e$ `in` C`;`	100%
	Operations and Operators on Multitudes		
15	forFromTo	**For** $i = 0,\ldots,n-1$ **[do]** ... `for` $(i$: e_1`..` $e_2)$ C`;` // $i = e_1,\ldots,e_2-1$ `for` $(i$: `..` $e)$ C`;` // $i = 0,1,\ldots,e-1$ `for` $(i$: e`..)` C`;` // $i = e,\ldots$ (∞-loop) `for (..)` C`;` // ∞-loop (no loop index)	73%
16	aggregate	**Aggregate** M **[st** $p(\cdot)$**] with** $A(\cdot,\cdot)$ M`.filter(...).reduce(...)`	58%
17	selectBy	**Select** $i \in M$ **s.t.** $p(i)$ $\{x \in M \mid p(x)\}$ $M/p(\cdot)$ M`.filter(...).collect(...)`	73%
18	forEach	**For** $i \in M$ **[s.t.** $p(i)$**]** **[do]** ... `for (..)` $p(\cdot)$`:`C M`.forEach(...)` M`.filter(...).forEach(...)`	96%
19	firstSuchThat	**First** $\cdot$ $M/p(\cdot)$ M`.filter(...).findFirst()`	50%

[a] e.g., pseudo-code, Java 8 streams, suggestion for extending plain Java syntax, etc.

[b] Prevalence score in the baseline corpus

Table 2: Baseline corpus of 26 OSS Java projects: *Identification*, *reproducibility* information and *work volume* indicators.

Project	First Version	Last Version	Last Hash	#Days	#Authors
Atmosphere[1]	10-04-30	14-04-28	557e1044	1,459	62
CraftBukkit	11-01-01	14-04-23	62ca8158	1,208	156
Cucumber-jvm	11-06-27	14-07-22	fd764318	1,120	93
Docx4j	12-05-12	14-07-04	8edaddfa	783	19
Elasticsearch	11-10-31	14-06-20	812972ab	963	129
Essentials	11-03-19	14-04-27	229ff9f0	1,134	67
Guava[1]	11-04-15	14-02-25	6fdaf506	1,047	12
Guice	07-12-19	14-07-01	76be88e8	2,386	17
Hadoop-common	11-08-25	14-08-21	0c648ba0	1,092	69
Hazelcast	09-07-21	14-07-05	3c4bc794	1,809	65
Hbase	12-05-26	14-01-30	c67682c8	613	25
Hector	10-12-05	14-05-28	f2fc542c	1,270	95
Hibernate-orm	09-07-07	14-07-02	b0a2ae9d	1,821	150
Jclouds	09-04-28	14-04-25	f1a0370b	1,823	100
Jna	11-06-22	14-07-07	a5942aaf	1,110	46
Junit[1]	07-12-07	14-05-03	56a03468	2,338	91
K-9	08-10-28	14-05-04	95f33c38	2,014	81
Lombok	09-10-14	14-07-01	3c4f6841	1,721	22
Mongo-java-driver	09-01-08	14-06-16	5565c46e	1,984	75
Netty	11-12-28	14-01-28	3061b154	762	72
Openmrs-core	10-08-16	14-06-18	05292d98	1,401	119
RxJava	13-01-23	14-04-25	12723ef0	456	47
Spring-framework	10-10-25	14-01-28	c5f908b1	1,190	53
Titan	13-01-04	14-04-17	55de01a3	468	17
Voldemort	01-01-01	14-04-28	fb3203f3	4,865	56
Wildfly	10-06-08	14-04-22	5a29c860	1,413	194
		Total:	*26 projects*	38,250	1,932

The complete study regarded a total of 78 Java projects, split into three parts: 26 belong GL Corpus, 26 are the most starred GitHub Java projects and the remaining 26 are the most starred Android GitHub projects at the time we collected the projects[6]. The GL corpus was used as a training corpus. At first the initial catalog of candidates was collected, through a process of *pattern harvesting*, by analyzing a set of six Java projects (partially overlapping with the GL corpus). Next, the prevalence of each nanos belonging the initial catalog of candidates, was computed on the GL corpus. Following this, the nanos whose prevalence is lower than the prevalence threshold, were discarded. Finally, the prevalence was computed for the projects of the remaining corpora (testing corpus). Information regarding the reproducibility appears in Table 2.

[6] February 2017

3 Lola

Lola combines computational expressiveness with minimal syntax. This is made possible through the adoption of Python as an underlying computation model. Lola also includes high level pattern matching, independence from the host language and a declarative nature (achieved with directives from C). Lola input is composed of *host tokens*–those of the host language, Python *snippets* of code, Lola *keywords* and *user defined* keywords. In order to distinguish the host-tokens from the Lola keyword (such as **`##Find`** and **`##replace`**), the former are defined in an XML configuration file.

The output stream is the result of the application of "directives" found in the input stream to subsequent input. The directives correspond to macros whose invocation is triggered by a pattern matching engine that relies on regular expressions over tokens. This makes it possible to augment the host language syntax without having to meddle with the language semantics. Macros are expanded to equivalent constructs written in the host language syntax. They can also be expanded in the augmented syntax, with the purpose of being further expanded by other Lola macros to code in the host language such as Java or C.

There are two kinds of directives: *Generator*s and *Lexie*s. Generators are constructs that use *Generating Expression*s (*GE*s) to return sequences of host-tokens inside the output stream. We can distinguish between two kinds of generators: *atomic* or *constructor*. Examples of atomic generators are **`##`**, **`##Include`** and **`##Import`**. Constructors are, for example, **`##If`**, **`##Unless`** and **`##Case`**. When the

preprocessor encounters generators, it pauses the process of copying host tokens from the input to the output stream and inserts into the stream the result of the generating expression.

Lexies are Lola's basic elements of computation and contain the instructions that determine the outcome of the Lola execution. In practice, they describe the augmentation to the host language. The augmentation is expressed in terms of *extended Regular Expressions* (*RE*s). When the preprocessor finds a lexi in the source code, it tries to match the *RE* reported in **##Find** directives against the subsequent tokens. Lexies are structured in sections, with a *Declarative Header*, an *Action Section*, and a *Declarative Footer*. Sections may occur in any order. Directives such as **##Find** and **##description** belong to the Declarative Header, which contains the *RE* that can be matched by the subsequent host language tokens. Section such as **##replace** and **##run** belong to the Action Section, which contains the elaborators for the code matching *RE*s in previous sections. The Declarative Footer usually contains directives such as **##example** and **##log**, used for documentation purposes.

*GE*s and *RE*s are both *Compound Object*s of Lola. GEs are used as conditional commands (i.e., **##If**, **##Unless**, etc.) or to iterate over the elements of a collection (usually a list) such as in the case of **##ForEach**. REs appears in a lexi after the **##Find** directive. When the input stream *RE* matches, Lola creates a Python object that contains information about the location in the input. This information is accessible by the other directive such as **##run** and **##replace** in other sections of the lexi. For example, it is possible to access the **self** and **__str__** attributes (and many others).

In sum, the Lola workflow is the following: Lola tries to match the pattern reported in the **##Find** directive. When it succeeds, this triggers, for example, the **##replace** directive reported in Fig. 4, which replaces the found pattern with code written in the host language. *Patterns* are extended regular expressions (REs). Whenever a snippet of code matches with a *RE* , Lola creates a Python *reifying* object, which can be manipulated. The computations performed by a lexi include code replacement, but a lexi can also invoke Python code.

Lola's syntax strives to adhere to English sentence structure. Furthermore, it uses capitalization conventions to make code easier to read: `@CamelCase` convention is used for constructors and `@camelCase` is for elaborators. The use of abbreviations or acronyms, including those of familiar terms such as EOF it is strongly discouraged. Lola can be used also for computing code metrics, enforcing code standards, and adding a C preprocessor functionality to any programming language. Nevertheless, its main purpose is to allow developers to introduce new keywords and operators, and in general, new syntax to the host language.

4 Case Studies

In the present Section we demonstrate the use of Lola to implement custom pluggable controllers with two specific goals:

1. To implement a Java Stenography.
2. To define new commands taken from a different programming language, Mathematica.

4.1 Stenography for Java Nano-Patterns

Stenography[7] is defined as *"an abbreviated symbolic writing method that increases speed and brevity of writing as compared to longhand, a more common method of writing a language"* [8]. The idea behind a stenography for Java nano-patterns, comes from the experience common to any developer. Everyone has written a lot of code is familiar with the feeling of repetitively writing small fragments of code. Although frequent fragments may take only few seconds to be written, if their frequency is high it may lead to a non-trivial effort for the developers, on the long run. With this mind, it is easy to understand that a stenography for nano-patterns is likely to increase programmers' productivity, by shortening the syntactic structure of several code constructs.

Nano-patterns are common Java idioms that have been proven to be recurring, namely prevalent, in a meaningful dataset of Java software systems. Since they refers to working code, their Lola implementation preserve the semantics in the host language. Table 1 provides the proposed stenography in the *Intent* column. Due to space constraints, we are going to illustrate two significant examples, one for each group. Specifically, for the first group we are illustrating notNullRequired nano whereas for the second group we are describing forEach.

Each snippet of code has a similar structure. The top part present a lexi, which reports the directives written in Lola language. These directives determine the substitutions of the tokens in the input stream. In the bottom part is reported the code governed by Lola. We can distinguish three kinds of keywords: *host specific* (Java keywords in this case), Lola's *built-in* (beginning with a double hash character **##**). These are *reserved* keywords. Finally we have *user defined* keywords, defined in Lola directives that, by convention, begin with a single hash character **#**. For example in Fig. 4 we have:

- **#safe** that is a new defined keyword.
- **##Find**, **##NoneOrMore**, **##Either**, **##or**, **##separator**, **##Any**, are Lola's keywords.
- **##ArgumentDeclaration**, **##Identifier**, **##SafeArgumentDeclarationList**, **##Type** are not primitive Lola's keywords, but are defined in a separate **##Find** directives contained in the **std.lola** library.
- **int** and **return** are Java keywords.

The first is an **##Import** directive:

##Import "std.lola" (4.1)

which is used to access to predefined directives. In fact, **##ArgumentDeclaration**, **##Identifier**,

[7] Also known as *shorthand writing*.

[8] https://en.wikipedia.org/wiki/Shorthand

##SafeArgumentDeclarationList, **##Type** are defined in the **std.lola** library.

The fundamental directive is **##Find**: It allows Lola to find sequences of tokens (patterns) in the input stream that match a specific extended regular expression (*RE*). For example in Fig. 5 the *RE* is the following:

$$for\ (\#\#Any(loop)\ |\ \#\#Expression(filter))\ \ \#\#Any(statements) \tag{4.2}$$

When a match is found Lola may trigger a number of actions, such as the substitution or deletion of elements, the recording of elements for later use, etc. The most common directive is the substitution according to the instruction found in the **##replace** directive (after the homonym reserved keyword). On occasion, it is possible to run Python script of code inside the **##run** directive. The mentioned Python code must be surrounded by curly braces.

Whenever a pattern matches, Lola creates a Python reifying object which can be manipulated in the **##run** section. The results of the matches are stored in variables whose identifiers correspond to the parameters of the directives. In case of multiple values, they are stored in lists.

For example, eq. (4.2) shows the use of an atomic *RE* , **##Any**. **##Any** is used twice one with **loop** parameter and the second with the **statements** parameter. **loop** and **statement** are the names of the variables in the reifying object. **##Expression** is a not a build keyword but is defined in the **std.lola** library.

4.1.1 The notNullRequired Nano

The notNullRequired is used to guard against **null** expressions. In case the expression is **null** the method returns. The lexi appears at the beginning of the code reported in Fig. 4. It introduces a user defined keyword called **safe**. The **#safe** modifier is applied to the parameters **x** and **s** of the method declaration at the bottom. It is used to check if the **##safe** parameters are **null**. In this case the method returns.

The first **##Find** directive defines the first pattern, which consists in a **#safe** modifier followed by the classic Java syntax for methods parameters. This is defined in the **##ArgumentDeclaration** directive in the imported **std.lola** library. Each match is stored in **SafeArgumentDeclaration**.

In the second **##Find**, *RE* matches if it finds:

1. one or more standard (unsafe) parameters declaration;
2. one or more **#safe** parameters declaration;
3. both the first and the second case interleaved (**##Either**);
4. none of them (**##NoneOrMore**).

In the third **##Find**, the *RE* matches the method declaration. Finally the **##replace** directive perform the substitution. It is worth to note the use of Python code, and specifically of the method **join** of the **String** class, inside a *list comprehension* expression, to generate the output. This script have access to **##SafeArgumentDeclaration** variable, to the element **l** of the **##SafeArgumentDeclarationList** **ls**. Each **l** has two fields (**name** and **type**, each of them has a **name** field).

```
int f(Integer x, String s) {
  Objects.requireNonNull(x);
  Objects.requireNonNull(s);
  return x + Integer.parse(s);
}
```

Figure 2: Java output of the notNullRequired nano

The Java implementation is reported in Fig. 2. It exploits the **java.util.Objects** class which includes 9 static methods to work on objects. For each method argument **x** with **#safe** modifier, adds a call to the static method **Objects.requireNonNull(x)** at the start of the method's body. This specific implementation throws **NullPointerException** when one of **#safe** arguments is **null**. However, we can think of other actions that can be done (i.e., printing a warning, returning null, etc.).

4.1.2 The forEach Nano

forEach implements a *filtered* **ForEach** loop. It is used to apply a command by iterating over a multitude or a subset of a multitude (as it happens in SQL). The lexi reported in Fig. 5 have a single **##Find** directive, that matches a pattern that includes an enhanced **for** loop and an expression separated by a *pipe* character **"|"**. The loop contains statements that are applied just if the expression after **"|"** is **true**.

```
for (final Integer i : nums()) {
  if (!(i%2==0))
  continue;
  f(i);
}
```

Figure 3: Java output of the forEach nano

Fig. 3 shows the Java code in output. The loop iterates over an Iterable returned by the nums() method, skipping the elements that do not satisfy the predicate *p* - in this case, if they are not even numbers. *p* can be a lambda expression.

4.2 Mathematica's Commands in Java

In the present section we present, as case study, the introduction of new commands in Java using Lola. We are particularly interested is on some commands of Mathematica, a popular language for technical computing. Mathematica presents several control structures that differs from those available in Java.

Table 3 reports the selection of the Mathematica's commands that we implemented. The second column reports the command in the Mathematica syntax, that is also the stenographic convention we adopted in the implementation. Descriptions are taken from the Mathematica documentation. Due to space constraints we are illustrating only one of the most meaningful commands, represented by the **Do** command.

```
##Import "std.lola"
##Find(SafeArgumentDeclaration) #safe ##ArgumentDeclaration
##Find(SafeArgumentDeclarationList) ##NoneOrMore ##Either ##ArgumentDeclaration(dec)
##or ##SafeArgumentDeclaration(dec) ##separator,
##Find ##Type(type) ##Identifier(name) (##SafeArgumentDeclarationList(ls)) {##Any(statements)}
##replace ##(type) ##(name) (##(",".join([l.type.name + "␣" + l.name.name for l in ls.decs]))) {
  ##("".join(["Objects.requireNonNull(" + l.name.name + ");" for l in ls.decs]))##(statements)}

int f(#safe Integer x, #safe String s) {
  return x + Integer.parse(s);
}
```

Figure 4: Java code governed by Lola for the notNullRequired nano

```
##Import "std.lola"
##Find for (##Any(loop) | ##Expression(filter)) { ##Any(statements) }
##replace for (##(loop)) {if (!(##(filter))) continue; ##(statements)}

for (final Integer i : nums() | i % 2 == 0) {
f(i);
}
```

Figure 5: Java code governed by Lola for the forEach nano

4.3 Loops: the Do Command

The **Do**[9] has multiple options and can be used in different ways. An example is reported in eq. (4.3): It iterates *n* times *expr*.

Do[expr, n] (4.3)

Fig. 6 reports the lexi and the Java code governed by Lola. It is possible to see the use of the **##run** directive: the **doName** variable is elaborated in the successive **##replace** directive as well as the **times** parameter of **##Literal** and the **ee** parameter of **##Any** in the **##Find** directive.

```
##Find do[##Any(e),{##Literal(times)}];
##run {
  if 'x' not in locals():
   x=0
  else:
   x=x+1
  doName='_i'+str(x)
}
##replace for(int ##(doName)=0;##(doName)<##(times);++##(doName)) {
  ##(e);
}

public void foo(){
  do[f(),{5}];
  do[g(),{15}];
}
```

Figure 6: Lexi for Do

Fig. 7 reports the output of the Lola engine. **f()** and **g()** are iterated *n* times (having *n* different values for each **for** loop).

```
public void foo(){
  for(int _i0=0; _i0<5; ++_i0) {f();}
  for(int _i1=0; _i1<15;++_i1) {g();}
}
```

Figure 7: Java output for Do

5 Discussion

Several opportunities may derive from the adoption of a pluggable, modular approach for programming language design. With every likelihood, some innovations in languages design might have been introduced faster and earlier. Also discussions around the new features would have been more sound, because supported by more realistic evidence of their feasibility and impact on languages architecture. In this work we are specifically interested in the design of control constructors. Lola is certainly helpful in this regards. Its preprocessing and macro language capabilities allow the final user to augment a language with new controllers (in a pluggable way) without affecting the language architecture.

With every likelihood using Lola we could have had **switch** on strings by augmenting the language with a library without changing the language version. It would have been the same for the C#'s **?.** operator[10]. Other improvements made possible by Lola is it multi-way, non-fall through branching conditional operator. And this would have been made using standard libraries of varieties, without changing language's core.

Some language extensions, such as the **?.** operator, are expected to be widely adopted by any kind of user, meaning that everyone is expected to eventually use them. Other application may be related to specific tasks. Programmers writing tests for an experts system may wish to define their own control constructors to support declarative tests such as

```
#Tweaking "int␣i=3;i+=2;" #gives "int␣i=5;";
```

Developers trained in the functional programming school, may find useful to use also in Java list expressions such as:

```
#sum #apply (_) -> 1./(_*_) #to primes();
```

Another possible application is represented by the development of libraries to deal with common tasks such as logging or interacting with SQL . As an example, consider Mockito, a popular "mocking framework for unit tests

[9] https://reference.wolfram.com/language/tutorial/LoopsAndControlStructures.html
[10] https://msdn.microsoft.com/en-us/library/dn986595.aspx

Table 3: Mathematica commands in Java

	Name	Command[a]	Description[b]
1	If	**If**[*condition*,*t*,*f*]	Gives *t* if *condition* evaluates to True, and *f* if it evaluates to False
		If[*condition*,*t*,*f*,*u*]	Gives *u* if *condition* evaluates to neither True nor False
2	Do	**Do**[*expr*,*n*]	Evaluates *expr* *n* times
		Do[*expr*,{*i*,i_{max}}]	Evaluates *expr* with the variable *i* successively taking on the values 1 Through i_{max} (in steps of 1)
		Do[*expr*,{*i*,i_{min},i_{max}}]	Starts with $i = i_{min}$
		Do[*expr*,{*i*,i_{min},i_{max},*di*}]	Uses steps di
		Do[*expr*,{*i*,{i_1,i_2,...}}]	Uses the successive values i_1, i_2, ...
		Do[*expr*,{*i*,i_{min},i_{max}},{*j*,j_{min},j_{max}},..]	Evaluates *expr* looping over different values of *j* etc., for each *i*
3	Nest	**Nest**[*f*,*expr*,*n*]	Gives an expression with *f* applied *n* times to *expr*
4	NestWhile	**NestWhile**[*f*,*expr*,*test*]	Starts with *expr*, then repeatedly applies *f* until applying test to the result no longer yields True
5	Which	**Which**[$test_1$,$value_1$,$test_2$,$value_2$,...]	Evaluates each of the $test_i$ in turn, returning the value of the $value_i$ corresponding to the first one that yields True

[a] Mathematica instruction.
[b] As reported in Mathematica's documentation.

in Java". Using a future (not the current) release of Lola, Mockito's developers would be able to rewrite the following snippet of code[11]:

```
Iterator i=mock(Iterator.class);
when(i.next()).thenReturn("Hello").thenReturn("World");
String result=i.next()+"␣"+i.next();
assertEquals("Hello␣World", result);
```

in the following way:

```
Iterator i=mock(Iterator.class);
#mock Iterator
#upon next() #return "Hello," #then "World!"
#affirm next() + "␣" + next() #is "Hello,␣World\n";
```

which is arguably more explanatory. With the present version of Lola it is not possible to implement the latest code, but it will be possible in the next release, currently under development.

In the Lola version, the following

```
#inlining...#to...
```

is a user defined control constructor, which can be seen as a syntactic sugar of the following instruction:

```
inliningInto("int␣i=3;i+=2;", "int␣i=5;");
```

In another form, using an appropriate fluent API library we have:

```
inlining("int␣i=3;i+=2;").to("int␣i=5;");
```

Lola can be seen as a special case of syntactic sugaring, task that is well performed by tools such as SugarJ [30], Racket [31] or Occam through Camlp4 [32]. Since Lola's focus is specifically on language extension it is possible to envision a rather coherent ensemble of applications of the idea of language extension, such as a definition of a DSL like fluent API. Lola can certainly be used during the DSLs development. At the same time, Lola itself can be seen as a DSL which uses *Macro Processing* as implementation approach [33].

Many other applications of Lola are possible in the field of testing, logging, design-by-contract, etc. (see, as an example, the recent work on *Seamless Requirement* from Naumchev and Meyer [34]) . Our current empirical study of nano-patterns in Java [2] indicates that nano-patterns occur in two thirds of methods, about half of the statements, third of conditional statements and 90% of all iterative statements, in the Gil-Lalouche corpus [29].

The basic scenario for Lola is that of language extension, amendment or augmentation. Users involved are developers, language engineers or advanced users interested in extending a General Purpose (GPL) or Domain Specific (DSL) host language. Apart from learning Lola's syntax (which, in the authors' opinion, should not have a steep learning curve for an average developer) the users need to know Python. This can be seen as a drawback of Lola. However, this issue is common to other similar solutions such as SugarJ [30], which presumes the knowledge of SDF [35] and Stratego [36]. On the other end, the use of Python might represent an opportunity, since its popularity [37,38], exposes Lola to a wider developers' community.

If language specification changes, this might in principle alter the behavior of the stenographic form of a nano-pattern imposed by Lola. In general, it is developers' responsibility to adjust Lola libraries according to the specification changes, so to guarantee *upward compatibility*. Nano-patterns have been proven to be popular solutions to small programming tasks, as results from their prevalence values. If we look at the baseline corpus described in Table 2, which represents a subset of the entire analyzed corpus, we find that the average *work volume* is, on average, 327.46 days (median 199 days), with a time range that spans from 2011 to 2014. During this time the Java Language Specification changed from version 3 (JLS3, 2004) [39] to 4 (JLS4, 2011) [40]. The JSL4 specification introduced important features, such as binary literals, diamond operators for generic type inference, etc.

Dyer et al., in their large scale investigation of Java projects hosted on SourceForge [12], found that new features are used by developers *before* the *official* release of the specification, taking advantage of the beta/pre-releases [41]. More likely than not, also the developers who worked on the projects included in the baseline corpus, adopted new language features during the development process. This means that the analysis is already, *implicitly*, taking into account the upward compatibility. In fact, the most prevalent nano-patterns emerged from source code that is likely to include both old and new features. This might suggest that either

[11] https://gojko.net/2009/10/23/mockito-in-six-easy-examples/
[12] https://sourceforge.net/

nano-patterns capture language constructs that were not interested by changes in the specification, or those that were affected by the mentioned changes, if present, did not pass our prevalence test.

Moreover, Dyer et al. found also that the majority of newly introduced language features are rarely used by developers [41], with few meaningful exception. According to Qiu et al. [42], who conducted a large-scale study on the use of Java constructs, the distribution of syntactic rules usage is *Zipfian*, with 20% of the most-used rules accounting for 85% of all rule usage, whereas the 65% of the least-used rules is used less than 5%. The same authors show that the adoption of new rules varies over time and it is contextual [42]. These findings might suggest that the problem of upward compatibility is not so significant in practice, being in fact mitigated by the tendency of developers to be, to a certain extent, recalcitrant to employ newly introduced features. Moreover, the decision to adopt an *h*-index variation as a reuse measure, is aimed at balancing the effect of such kind of statistical distribution, as discussed in Sect. 2.2.

6 Related Work

6.1 Lola

6.1.1 Embedded languages

Lola can be seen as an *embedded language*. The embedded text is distinguished by the host text because it is preceded by a hash character and ends with an un-escaped end-of-line character. The most trivial example involves comments. To a certain extent, also comments and string literals can be seen as embedded language. In Java, for example, comments begin and end with specific character sequences, `/*` and `*/`, respectively for the beginning and end of the comment. The same applies to JavaDoc comments. Inside a comment, text is treated differently than in regular code.

The most relevant example is represented by PHP, where commands are embedded in HTML [43] pages. Other meaningful examples are ASP [44] and JSP [45], which adopted the same idea. More recently we have a number examples of DSL languages which extend or enhance a host GPL language. They usually target specific problem, like interaction with databases as in the case of J% [46].

Lola is not an embedded language on its own. It actually embeds both the host language and Python. The host language is seen as a stream that can be manipulated, whereas Python snippets are used to compute the code processing.

6.1.2 Preprocessors

There are several preprocessors available. The C preprocessor [47], introduced in the early stages of its host language, C, is arguably the most famous text preprocessor, often identified by its acronym, CPP [48, 49]. It is almost entirely independent from the host programming language [50], to the point that it can be viewed as an independent programming language. As a programming language, the C preprocessor is limited. For example, it cannot perform iterations (loops) and it does not have conditionals on macro parameters and it does not allow recursive structure. It also has a minimal "library".

Other popular preprocessors are PL/1 and M4 [51]. The PL/1 preprocessor is similar to the C preprocessor in some respects (i.e., built-in types, directives, etc.) whereas M4 is a general language-independent preprocessor, which facilitates more complete processing (it allows recursive calls and has conditional constructs) if compared to the other above mentioned [52] preprocessors. M5 [53] is an improved version of M4.

Many preprocessors use a general purpose programming language, such as Python (e.g., Cog[13], jinja[14], django[15], PYM [54], pyexpander[16], pypreprocessor[17], Preprocessor in Python[18] and Cheetah[19]). Others use Perl (e.g., PerlPP[20]).

6.1.3 Macro Languages

There are several problems with macro systems. They might capture names that are already used (Hygenic Macros solve this problem [55]). Macro systems do not usually differentiate between lexical elements of the hosting language such as expressions, identifiers, constants, etc. One preprocessor that differs in this aspect is the Marco preprocessor [56], which has a way to reduce the coupling between the host language to the macro system.

6.2 Nano-Patterns

The first appearance of the term nano-patterns was in the conclusion of a work by Gil and Maman micro-patterns [57]. Singer et al. [19], starting from a work by Høst and Østvold [58], collected a language of 17 nano-patterns, which the authors demonstrated to be *prevalent* (at an 80% level), *traceable*, and, *purposeful*. The names of their found nano-patterns underscore their purposefulness. Singer et al.'s catalog is at the base of studies on the relationship between nano-patterns and defectiveness [59] or vulnerabilities [60].

Differently from our catalog, Singer et al.'s catalog regards properties of methods, whereas the nanos in our catalog are found at the method, command, expression and field

[13] `http://nedbatchelder.com/code/cog/`, Cog by Ned Batchelder.
[14] `http://jinja.pocoo.org/`, jinja by Armin Ronacher.
[15] `https://docs.djangoproject.com/en/1.7/topics/templates/`, Django Software Foundation.
[16] `http://pyexpander.sourceforge.net/`, pyexpander by Goetz Pfeiffer.
[17] `https://code.google.com/p/pypreprocessor/`, pypreprocessor by Evan Plaice.
[18] `http://orbeckst.github.io/GromacsWrapper/gromacs/core/fileformats/preprocessor.html`, Preprocessor in Python by Evan Plaice and Oliver Beckstein.
[19] `http://www.cheetahtemplate.org/`, Cheetah by Tavis Rudd.
[20] `https://github.com/d-ash/perlpp`, PerlPP by Andrew Shubin.

levels [2]. Singer et al.'s nanos are orthogonal. Each nano-pattern represents a binary property, meaning that it can be present or not. This property lead to the definition of composite nano-patterns, which are those that are a combination of *fundamental* nano-patterns.

Other works on nano-patterns are those of Batarseh [20], who presents a language formed by 16 method properties and Lee et al. [61], who report 67 attributes of different kind of nanos: method signature, body and ties of body and behavior.

6.2.1 Other Kinds of Patterns

In the software engineering literature (scientific and not), there has been a surge of interest regarding the theory and application of "design patterns". Such patterns are a specific arrangement of object oriented components (classes, methods, inheritance) and represent recurring solutions to common design problems [62, 63]. Research interest revolves, for example, around design patterns automatic detection.

At a lower level of granularity stands the "micro patterns" (or μ-patterns), which are predicated on OO types (classes). Micro patterns reflect a specific use of OO features, such as the absence of methods, inheritance, etc. [57]. Differently from design patterns, the 27 micro patterns in Gil and Maman's catalog are *prevalent*, which means that they are present in around 75% of all classes, as empirical studies have shown [57]. Micro patterns are *traceable*, in the sense that they can be automatically recognized. In opposition, design patterns are not traceable [64] and many attempts have been made to formalize [65–69] and to automatically detect them [70–75].

Design patterns are also purposeful, which means that they deal with a specific problem. This is not the case of μ-patterns (they just track the presence of a coding technique), albeit the prevalence might suggest that a μ-pattern is used on purpose. The lack of purpose is also a characteristic of nano-patterns, although for a different reason. Their purpose is related to the small programming task that they carry out, and can often be learned from their name, but it is usually unrelated to the system where they occur.

6.2.2 Other Pattern-Like Constructs

A concept similar to nano-patterns is that of *idiom*s, which are, quoting Allamanis and Sutton, "a syntactic fragment that recurs frequently across software projects and has a single semantic purpose" [76]. The main difference between idioms and nano-patterns is that the latter are not single fragments but predicates.

Notably we can find idiom catalogs for different languages, e.g., [77–80]. Also, IDEs provide support for the definition and customization of idioms [81, 82]. Some instructional/educational material on idioms is found in work whose aim is to teach idioms as programming tips [83–85].

Sutton et al. investigated the presence of idioms in generic C++ libraries, finding a high coverage (circa 85% of classes showing idioms) [86]. In a recent work, Allamanis and Sutton applied machine learning techniques to *automatically* detect idioms in source code. The discovered idioms included "cross-projects idioms that represent important program concepts like object creation, exception handling, etc." with a prevalence ranging between 3% to 31%, depending on some factors (i.e., the training and testing dataset, the used parameters, etc.) From their work, we can see that the prevalence of idioms tends be lower than that of nanos [76].

Another difference between the two constructs is that nanos are *sought*, whereas idioms are discovered. Some of the found idioms show semantic purposes, in the sense that they are used for object creation, exception handling, and resource management. In the literature on idioms, other relevant studies are those of Koening [87], Langer [88] and Willis [89].

6.2.3 Vocabulary *vs.* Structure

Vocabulary: Some related work deals with code nomenclature or the study of the vocabulary that developers use to describe program elements. Linstead et al. studied source code vocabulary and discovered the existence of naming trends related to specific syntactic elements such as classes and interfaces [90]. Enslen et al. worked on an optimized algorithm to split identifiers into words' sequences [91]. Abebe et al. studied the evolution of vocabulary used by developers along a time line [92]. Newman et al. studied how to determine and assign lexical categories starting from source code [93].

Høst and Østvold investigated the implementation of methods from a corpus of Java applications, to determine which word is the best for a method naming [94]. In a subsequent work, they dealt with generation of *"a semantics which captures our common interpretation of method names"* [58]. They worked on *traceable pattern*s and proposed a set of *traceable attributes* that they claim can be useful building blocks of nano-patterns.

It is worth noting that the authors *explicitly* claim they were not proposing nano-patterns. The way vocabulary matches the structure or not can also lead to "naming bugs", a problem that can be mitigated by an automatic procedure. Høst and Østvold proposed automatic tool [95] to suggest proper names. Kashiwabara et al., in different works, presented techniques to identify candidate verbs for methods [96, 97]. There were some attempts to use recurring *structure* for software engineering purposes. Examples are works on *beacons* [98, 99]—stereotypical, recurring segments of code that are quickly recognized by experienced developers.

Structure: Some researchers in the area focus their efforts on investigating the recurring elements of the syntactical *structure* of programs. A concept similar to nano-patterns is that of *stereotype*. A stereotype is a syntactical structure that capture the intent of methods and classes and can be considered an extension of the micro-nano-pattern concept [100].

Andras et al. and Moreno et al. compared the outcome of the run-time with their stereotype [101], to investigate the consistency between method design and implementation [101, 102]. Qiu et al. showed that the use of syntacti-

cal rules in actual programs follows a Zipf-like distribution (just as happens with words in natural language). In other words, small sets of syntactic structures tend to govern entire projects [103]. Abd-El-Hafiz et al. classified loops by complexity levels [104]. Wang et al. introduced an automatic approach to determine high level action through the analysis of a given loop [105]. Mens et al. proposed to describe patterns with a declarative meta-programming language similar to Prolog.

7 Conclusion

We introduced the concept of pluggable controllers as a way to facilitate the introduction of new constructors, and explained how they can be implemented using Lola, the Language of Language Amendment, a powerful preprocessor and macro-language. Lola lets developers augment or even amend language constructors without affecting the language architecture.

This argument was illustrated by two cases study: A stenography for Java nanos and the introduction of Mathematica's commands in Java. Nanos are recurring idioms devised by developers to perform simple tasks. In a recent study they revealed to be prevalent in Java, under a given definition of prevalence. We presented 19 nanos that belong to a wider catalog [2] and illustrated the implementation of Java stenography based on them.

Their prevalence make them good candidates to become pluggable controllers. A stenography for nanos, such that illustrated in this work for Java—and its subsequent definition in standard libraries of pluggable controllers, is likely to improve developers efficiency by reducing coding effort.

In the present paper we reported a peculiar application of Lola. However, Lola's applications are not limited to the extension of SIC as pluggable controllers. For example, with Lola it is possible to compute the Halstead metrics [106], simulate Aspect Oriented Programming, add syntactic sugar, and many other kinds of language augmentation [15].

Relying on a powerful preprocessing engine, Lola makes it easy to introduce augmentation. Formal and precise proofs of semantics are however difficult: Since the semantics of the underlying language is not available to preprocessors, even the formulation of precise statements on semantics seems impossible. More so, with Lola which is language independent, and draws much of its power from the loose coupling with the underlying semantics and changes made to it.

7.1 Future Work on Lola

Future works will involve both further improvement of Lola and research on the application of Lola to help developers' work.

- Currently, Lola supports *trivia* (elements such as space, tabs, etc.), though it is limited to just a few of them such as `##NewLine` and `##EndOfFile`, and must be improved. This type of support may be needed to process comments, JavaDoc (for Java) and the like.
- Pattern matching features should be improved in order to treat string literals. The introduction of features that allow the host language to be changed on the fly would enable application of Lola's preprocessing to mixed code (i.e., Java code with an SQL queries as happens using JDBC, etc.)
- Another promising research avenue emerges if we ponder the possibility of applying Lola to itself.
- We would like to allow users to define new Lola keywords from within Lola itself (now only possible with simple Python classes).
- Some enhancement are required to improve usability. For example, using single and double # to distinguish Lola keywords from host language identifiers might lead to some confusion. We are working on a different solution, such as using another character (e.g., @) instead of #.

7.2 Future Work on Nanos

Further research it is needed to improve the prevalence values of nano-patterns by specifically looking for new candidates in other corpora. In this work, we used a lightweight nano tracer to track the nanos in source code. Additional work it is needed to improve search possibilities in several aspects (i.e., improving the namespace analysis, etc.). The subjective and sample bias in the harvest may have led us to miss some patterns. Further efforts are needed to verify this suspicion and to extend the basic catalog.

Conflict of Interest The authors declare no conflict of interest.

Acknowledgment The authors thank the anonymous reviewers for their valuable suggestions. Inspiring discussions with Tomer Levy are gratefully and intentionally acknowledged. This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 1803/13 *).

References

[1] Yossi Gil, Ori Marcovitch, and Matteo Orrù. Pluggable controllers and nano-patterns. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 447–451. IEEE Computer Society, 2017.

[2] Yossi Gil, Ori Marcovitch, and Matteo Orrù. A Nano-Patterns Language for Java. Draft available at https://-goo.gl/tSte12, 2017.

[3] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004.

[4] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[5] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[6] Donald E. Knuth. Structured programming with Go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974. ACM, New York, NY, USA.

[7] H.D. Mills. Mathematical foundations for structured programming. Technical report, IBM rep. FSC 72-6012, IBM Fed. Syst. Div., Gaithersburg, Md., 1972.

[8] Niklaus Wirth. On the composition of well-structured programs. *ACM Comput. Surv.*, 6(4):247–259, December 1974. ACM,New York City, NY, USA.

[9] J. R. Donaldson. Structured programming. In Edward Nash Yourdon, editor, *Classics in Software Engineering*, pages 179–185. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[10] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, may 1966. ACM,New York City, NY, USA.

[11] Bertrand Meyer. *EIFFEL the Language*. Object-Oriented Series. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[12] Walter Cazzola and Edoardo Vacchi. Language components for modular DSLs using traits. *Computer Languages, Systems & Structures*, 45:16–34, apr 2016. Springer Publishing, New York City, NY, USA.

[13] Nathanael Schrli, Stéphane Ducasse, and Oscar Nierstrasz. Traits: Composable units of behavior. In *(ECOOP'03)*, volume 2743, pages 248–274, 2003-07.

[14] Leonardo V.S. Reis, Vladimir O. Di Iorio, and Roberto S. Bigonha. An on-the-fly grammar modification mechanism for composing and defining extensible languages. *Computer Languages, Systems & Structures*, 42:46–59, jul 2015. Springer Publishing, New York City, NY, USA.

[15] Iddo E. Zmiry. Lola: a Programming Language for Augmenting Programming Languages. Master's thesis, Technion—Israel Institute of Technology, 2016. url: https://drive.google.com/file/d/0B3645jTHku6WZzVkMl9uVGlTQ2M/view.

[16] Dennis M. Ritchie. The evolution of the UNIX time-sharing system. In *Language Design and Programming Methodology*, pages 25–35. Springer, 1980.

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

[18] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 97–116, New York, NY, USA, 2005. ACM.

[19] Jeremy Singer, Gavin Brown, Mikel Luján, Adam Pocock, and Paraskevas Yiapanis. Fundamental nano-patterns to characterize and classify java methods. *Elect. Notes Theor. Comp. Sci.*, 253(7):191–204, September 2010. Springer Publishing, New York City, NY, USA.

[20] Feras Batarseh. Java nano patterns: A set of reusable objects. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 60:1–60:4, New York, NY, USA, 2010. ACM.

[21] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proc. National Academy of Sciences of the USA of America*, 102(46):16569–16572, 2005. National Academy of Sciences.

[22] Leo Egghe and Ronald Rousseau. An infometric model for the Hirsch-index. *Scientometrics*, 69(1):121–129, 2006. Springer Publishing, New York City, NY, USA.

[23] Tal Cohen and Joseph Gil. Self-calibration of metrics of Java methods. In *TOOLS Pacific'00: 37th Int. Conf. Tech. OO Lang. & Syst.*, pages 94–107, Washington, DC, 2000. IEEE. IEEE,Washington, DC, USA.

[24] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large oo soft. syst. *IEEE*, 33(10):687–708, October 2007.

[25] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu. On the distribution of bugs in the eclipse syst. *IEEE*, 37(6):872–877, November 2011.

[26] I. Turnu, G. Concas, M. Marchesi, S. Pinna, and R. Tonelli. A modified yule process to model the evolution of some OO system properties. *Inf. Sc.*, 181(4):883–902, 2011. Springer Publishing, New York City, NY, USA.

[27] S. Alonso, F.J. Cabrerizo, E. Herrera-Viedma, and F. Herrera. h-index: A review focused in its variants, computation and standardization for different scientific fields. *Journal of Informetrics*, 3(4):273 – 289, 2009.

[28] Miltiadis Allamanis and Charles A. Sutton. Mining source code repositories at massive scale using language modeling. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Proc. the 10th Working Conf. Mining Soft. Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216, NY/USA, 2013. IEEE.

[29] Joseph (Yossi) Gil and Gal Lalouche. When do soft. complexity metrics mean nothing?—when examined out of context. *J. Object Tech.*, 15(1):2:1–25, 2016.

[30] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, oct 2011. ACM, New York, NY, USA.

[31] Matthew Flatt. Creating languages in Racket. *Communications of the ACM*, 55(1):48–56, January 2012. ACM,New York City, NY, USA.

[32] Daniel de Rauglaudre. Camlp4–reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, 2003. Accessed: 2016-08-26.

[33] Tomaž Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, apr 2008.

[34] A. Naumchev and B. Meyer. Seamless requirements. *Computer Languages, Systems & Structures*, 49:119 – 132, 2017. Springer Publishing, New York City, NY, USA.

[35] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, November 1989.

[36] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52 – 70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[37] PYPL PopularitY of Programming Language. `http://pypl.github.io/PYPL.html`. Accessed: 2017-07-31.

[38] TIOBE Index for July 2017. `https://www.tiobe.com/tiobe-index/`. Accessed: 2017-07-31.

[39] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[40] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[41] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

[42] Dong Qiu, Bixin Li, Earl T. Barr, and Zhendong Su. Understanding the syntactic rule usage in java. *Journal of Systems and Software*, 123:160 – 172, 2017.

[43] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C Recommendation, W3C - World Wide Web Consortium, December 1999.

[44] George Reilly Brian Francis and Dino Esposito. *Professional Active Server Pages 3.0*. Wrox Press, 1999-09.

[45] Hans Bergsten. *JavaServer Pages*. O'Reilly Media, 3rd edition, 2003-12.

[46] Vassilios Karakoidas, Dimitris Mitropoulos, Panagiotis Louridas, and Diomidis Spinellis. A type-safe embedding of SQL into Java using the extensible compiler framework J%. *Computer Languages, Systems & Structures*, 41:1–20, apr 2015. Elsevier Science Publishers, Amsterdam, The Netherlands.

[47] GNU. C preprocessor. `https://gcc.gnu.org/onlinedocs/cpp/`.

[48] Dennis Ritchie. he Development of the C Language. In John A. N. Lee and Jean E. Sammet, editors, *HOPL Preprints*, pages 201–208. ACM, 1993.

[49] A. Snyder. A portable compiler for the language C. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[50] GNU. The C preprocessor – overview. `https://gcc.gnu.org/onlinedocs/cpp/Overview.html#Overview`.

[51] GNU. M4 manual. `http://www.gnu.org/software/m4/manual/m4.html`.

[52] GNU. M4 manual – history section. `https://www.gnu.org/software/m4/manual/m4.html#History`.

[53] A. Dain Samples. *User's Guide to the M5 Macro Language*. Number UCB/CSD-91-621 in Report // Computer Science Division (EECS), UCB/CSD. University of California, Berkeley, Computer Science Division, 2nd edition, 1991.

[54] Robert F Tobler. Pym-a macro preprocessor based on python. In *Proceedings of the 9th International Python Conference, Long Beach, California*, 2001.

[55] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on Lisp and functional programming*, pages 151–161, 1986. ACM, New York, NY, USA.

[56] Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S McKinley. Marco: Safe, expressive macros for any language. In *ECOOP 2012–Object-Oriented Programming*, pages 589–613. Springer, 2012.

[57] Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In Ralph Johnson and Richard P. Gabriel, editors, *Proc. the 20th Annual OO Prog., Syst., Lang., and App., OOPSLA 2005*, volume 40, pages 97–116, NY/USA, October 2005. ACM.

[58] Einar W. Høst and Bjarte M. Østvold. The java prog.'s phrase book. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Soft. Language Eng., First Int. Conf., SLE 2008, Toulouse, France, Sep 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Comp. Science*, pages 322–341, NY/USA, 2008. Springer.

[59] A.K. Deo and B.J. Williams. Preliminary study on assessing software defects using nano-pattern detection. In *24th International Conference on Software Engineering and Data Engineering, SEDE 2015*, 2015.

[60] K. Z. Sultana, A. Deo, and B. J. Williams. A preliminary study examining relationships between nano-patterns and software security vulnerabilities. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 257–262, June 2016. IEEE,Washington, DC, USA.

[61] Illo Lee, Suntae Kim, Sooyong Park, and Younghwa Cho. *Attributes for Characterizing Java Methods*, pages 185–191. Springer, Berlin, Heidelberg, 2016.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[63] Erich Gamma. Going beyond objects with design patterns. In *ECOOP'97—OO Prog.: 11th European Conf. Jyväskylä, Finland, Jun 9–13, 1997 Proc.*, pages 530–530, Berlin, Heidelberg, 1997. Springer.

[64] Peter Van Emde Boas. Resistance is Futile; formal linguistic observations on design patterns. techreport ILLC-CT-1997-03, The Institute For Logic, Language, and Computation (ILLC), University of Amsterdam, 1997-02.

[65] Amnon H. Eden, Joseph (Yossi) Gil, Yoram Hirshfeld, and Amiram Yehudai. Motifs in object oriented architecture, 1999.

[66] Amnon H. Eden, Joseph (Yossi) Gil, and Amiram Yehudai. A formal language for design patterns. In *(PLoP'96)*, 1996-09.

[67] Amnon H. Eden. Formal specification of object-oriented design. In *Int. Conf. on Multidisciplinary Design in Engineering (CSME-MDE'01)*, pages 21–22, 2001-11.

[68] Amnon H. Eden. A visual formalism for object-oriented architecture. In *(IDPT'02)*, 2002-06.

[69] Amnon H. Eden. Giving "the quality" a name - precise specification of design patterns - a second look at the manuscripts. *Journal of Object-Oriented Programming*, 11(3), 1998-06.

[70] Rudolf Ferenc, Árpád Beszédes, Lajos Jeno Fülöp, and Janos Lele. Design pattern mining enhanced by machine learning. In *21st "IEEE" 2005), 25-30 Sep 2005, Budapest, Hungary*, pages 295–304, NY/USA, 2005. IEEE.

[71] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, and Houari Sahraoui. Improving design-pattern identification: a new approach & an exploratory study. *Soft. Quality J.*, 18(1):145–174, 2010. Kluwer Academic Publishers, Hingham, MA, USA.

[72] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *J. Syst. and Soft.*, 103:102–117, 2015. Springer Publishing, New York City, NY, USA.

[73] Yann-Gaël Guéhéneuc, Houari A. Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *11th Working Conf. Rev. Eng., WCRE 2004, Delft, The Netherlands, Nov 8-12, 2004*, pages 172–181, NY/USA, 2004. IEEE.

[74] Marek Vokác. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5(1):139–157, 2006.

[75] Linzhang Wang, Zhixiong Han, Jiantao He, Hanfei Wang, and Xuandong Li. Recovering design patterns to support program comprehension. In *Proceedings of the 2Nd International Workshop on Evidential Assessment of Software Technologies*, EAST '12, pages 49–54, New York, NY, USA, 2012. ACM.

[76] Miltiadis Allamanis and Charles A. Sutton. Mining idioms from source code. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proc. the 22nd "ACM" Found. Soft. Eng., (FSE-22)*, pages 472–483, Hong Kong, November 2014. ACM.

[77] Wikibooks. More C++ Idioms, 2017. https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms.

[78] Java Idioms Editor. Java Idioms, 2017. http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html.

[79] S. Chuan. JavaScript Patterns Collection, 2014. http://shichuan.github.io/javascript-patterns/.

[80] R. Waldron. Principles of writing consistent, idiomatic JavaScript, 2014. https://github.com/rwaldron/idiomatic.js/.

[81] E. Recommenders-Contributors. Eclipse SnipMatch, 2014.

[82] JetBrains. High-speed coding with Custom Live Templates., 2014. http://bit.ly/1o8R8Do.

[83] Nathan Gurewich and Ori Gurewich. *Java Manual of Style*. Ziff-Davis Publishing Co., April 1996.

[84] Chris Laffra. *Advanced Java: Idioms, Pitfalls, Styles & Prog. Tips*. Prentice Hall Ptr, September 1996.

[85] Craig Larman and Rhett Guthrie. *Java 2 Performance & Idiom Guide*. Prentice Hall Ptr, NJ/USA, 1999.

[86] A. Sutton, R. Holeman, and J. I. Maletic. Identification of idiom usage in c++ generic libraries. In *2010 IEEE 18th Int. Conf. Program Comprehension*, pages 160–169, New York,NY, June 2010. IEEE.

[87] Andrew Koenig. Idiomatic design. *Comm. ACM*, pages 14–19, 1995. ACM, New York, NY, USA.

[88] A. Langer. Java programming idioms. In *Proc. OO Lang. & Syst.. TOOLS 38*, pages 197–198, 2001. IEEE,Washington, DC, USA.

[89] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artif. Intell.*, 45(1-2):113–171, September 1990. Elsevier Science Publishers Ltd. Essex, UK.

[90] Erik Linstead, Lindsey Hughes, Cristina Lopes, and Pierre Baldi. Exploring Java software vocabulary: A search & mining perspective. In *Proc. Int. Conf. Soft. Eng.*, pages 29–32, 2009. IEEE,Washington, DC, USA.

[91] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 71–80, Washington, DC, USA, May 2009. IEEE.

[92] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 189–198, Washington, DC, USA, March 2009. IEEE.

[93] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, and J. I. Maletic. Lexical categories for source code identifiers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 228–239, Feb 2017. IEEE,Washington, DC, USA.

[94] E. W. Host and B. M. Ostvold. The programmer's lexicon, volume i: The verbs. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, Sept 2007. IEEE, Washington, DC, USA.

[95] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[96] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, and K. Inoue. Recommending verbs for rename method using association rule mining. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 323–327, Feb 2014. IEEE,Washington, DC, USA.

[97] Yuki Kashiwabara, Takashi Ishio, Hideaki Hata, and Katsuro Inoue. Method verb recommendation using association rule mining in a set of existing projects. *IEICE Transactions on Information and Systems*, E98.D(3):627–636, 2015.

[98] Ruven Brooks. Towards a theory of the comprehension of computer programs. *Int. J. Man-Machine Studies*, 18(6):543–554, 1983. Elsevier Science Publishers, Amsterdam, The Netherlands.

[99] Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice & expert programmers. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *Proc. 14th Ann. Workshop of the Psychology of Programmers Interest Group (PPIG)*, pages 58–73, 2002.

[100] N. Dragan, M. L. Collard, and J. I. Maletic. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 24–34, Washington, DC, USA, Sept 2006. IEEE.

[101] P. Andras, A. Pakhira, L. Moreno, and A. Marcus. A measure to assess the behavior of method stereotypes in OO software. In *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 7–13, Washington, DC, USA, May 2013. IEEE.

[102] L. Moreno and A. Marcus. Jstereocode: automatically identifying method and class stereotypes in java code. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 358–361, New York, NY, Sept 2012. IEEE.

[103] Dong Qiu, Bixin Li, Earl T. Barr, and Zhendong Su. Understanding the syntactic rule usage in Java. *J. Syst. and Soft.*, 123:160–172, 2017. Springer Publishing, New York City, NY, USA.

[104] S. K. Abd-El-Hafiz and V. R. Basili. A knowledge-based approach to the analysis of loops. *IEEE*, 22(5):339–360, May 1996.

[105] X. Wang, L. Pollock, and K. Vijay-Shanker. Developing a model of loop actions by mining loop characteristics from a large code corpus. In *2015 IEEE 31st Int. Conf. Soft. Maint. & Evolution, ICSME 2015 - Proc.*, 2015. IEEE,Washington, DC, USA.

[106] Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems. Elsevier Science Inc., New York, NY, USA, 1977.