A Nano Patterns Language for Java

Ori Marcovitch
A Nano Patterns Language for Java

RESEARCH THESIS

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Ori Marcovitch

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5777, Haifa, April 2017
This Research Thesis was Done under the Supervision of Prof. Yossi Gil in the Faculty of Computer Science

Some results in this thesis have been published as an article by the author and research collaborators in a conference during the course of the author’s research period, the most up-to-date version of which being: Gil et al. (2017). The results were also submitted to OOPSLA (2017).

Acknowledgments

The Generous Financial Help of the Technion is Gratefully Acknowledged
Contents

Abstract 1

1 Introduction 3
   1.1 Parable .................................................. 3
   1.2 Nano Patterns and Language ............................. 4
   1.3 Evaluation ................................................. 7
   1.4 The Prevalence Threshold ............................... 8
   Outline ...................................................... 9

2 Context and Applications 11

3 The Nanos Language 15

4 Language Prevalence 23

5 Language Evolution and Implementation 27
   5.1 Phase 0: Candidate Collection .......................... 28
   5.2 Phase I: Candidate Pruning .............................. 29
   5.3 Phase II: Candidates to Nanos .......................... 31

6 The Prevalence Threshold 33

7 Related Work 37
   7.1 Design Patterns and Micro Patterns .......................... 37
   7.2 Automatic Discoverable Idioms ............................ 38
   7.3 Previous Work on Nano Patterns ........................... 39

8 Threats to Validity and Critique 41
   8.1 Limited Domain ........................................... 41
   8.2 Missed Nanos and the Language $\mathcal{J}$ ................. 42
# Nanos Detection Analysis

9.1 The Spartanizer .................................................. 45
9.2 Leonidas ............................................................... 46
   9.2.1 Motivation .................................................... 46
   9.2.2 Solution ......................................................... 46

## Conclusion

10 Conclusion .................................................................. 49

## Other Related Work

A Other Related Work .................................................. 51

## Nano Snippets

B Nano Snippets ............................................................ 55
   B.1 Elaborators ......................................................... 55
   B.2 Multitude ............................................................. 57
   B.3 Phrase ................................................................. 58
   B.4 Field ................................................................. 58
   B.5 Sink Method ......................................................... 59
   B.6 Delegating Method ............................................... 60
   B.7 Constructor ........................................................ 61

## Reproducibility Information

C Reproducibility Information ........................................ 63

## Java Like Stenography

D Java Like Stenography ................................................. 69

## Hebrew Abstract

Hebrew Abstract ................................................................ 80
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Language of Java Nano Patterns</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Aggregating statistics of prevalence (in percent point) of the nanos language in three corpora of 26 OSS projects each.</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Corpus of 6 OSS Java projects used for preliminary testing: identification, reproducibility information and size metrics.</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Nano Candidates</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Candidates phrased as nanos, but rejected later</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>Summary statistics of r-index values in the baseline corpus (Table 11)</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>The Leonidas language</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>Baseline corpus of 26 OSS Java projects: identification, reproducibility information and work volume indicators.</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>Reproducibility information for the Java corpus consisting of the 26 most starred Android projects on GitHub</td>
<td>65</td>
</tr>
<tr>
<td>10</td>
<td>Reproducibility information for Android corpus consisting of the 26 most starred Android projects on GitHub</td>
<td>66</td>
</tr>
<tr>
<td>11</td>
<td>Values of the $r$-, $r_i$-, and $r_x$- indices in the baseline-corpus (Table 8) at the end of the monitored period, and differences from initial values</td>
<td>67</td>
</tr>
<tr>
<td>12</td>
<td>The 5 sub-nanos augmenting the language of Table 1</td>
<td>68</td>
</tr>
</tbody>
</table>
List of Figures

1. Taxonomy of the language of Java nanos . . . . . . . . . . . . 5
2. Reproduction of Fig. 1 . . . . . . . . . . . . . . . . . . . . . 15
3. Class for tracing the \texttt{defaultsTo} in Java code . . . . . . . . . . . . 28
Abstract

Patterns, micro-patterns, and nano-patterns have many applications: program comprehension, code transformations, documentation aides, improving code robustness, etc. This work revisits nano-patterns, taken here to be predicates on short code snippets that represent some common and elementary programming missions such as “for each \( m \in M \) do ...”, or, “use \( x \) (but if \( x \) is \texttt{null}, \( y \) is a substitute)”.

The thesis describes a taxonomized language of 38 nano-patterns for Java. We also describe the process of patterns harvesting we used and the underlying rationale, including the prevalence threshold criterion, which, capitalizing on Hirsch’s famous h-index, makes a robust yard stick of pattern’s significance.

An empirical survey of 78 OSS Java projects indicates that the language has substantial prevalence in code: about a third of commands (executable statements) and half of methods are instances of nano patterns in the proposed language. Also, the language’s prevalence is typically higher than that of languages harvested in a project specific, automated machine learning process.

Nano-patterns are implementation/language level details for most high level software engineering purposes. A contribution here is in identifying this clutter, appreciating its presence, and imposing order on it. The language, the nano patterns in it, and the contributed automatic tool for tracing nano patterns in code, may help to deal systematically with this low-level, yet significant, portion of code.
Chapter 1

Introduction

1.1 Parable

Start with this parable. A team of professionals designs a snacks’ vending machine. Mike, the mechanical engineer, is responsible of mechanical aspects including these sub-systems of the vending machine: buttons, slots, the coin mechanics, motors, etc. To attach these modules, Mike needs nuts and bolts of many kinds. Mike can order (or make) custom made nuts, bolts, and even threaded rods in any desired handedness, pitch and diameter. Mike is skilled at making these custom made selections. He is well trained in appreciating the relative merits of the parameters that define these. He may even use custom made nuts and bolts, e.g., for securing the safe and lock. Still, productivity sake dictates that most nuts and bolts should be ordered from some standard set of preferred combination of diameter and pitch, such as ISO 262 (ISO, 1998).

In choosing to use ISO 262 nuts and bolts, Mike reuses the design decisions of the ISO committee. Mike’s reuse is not limited to nuts and bolts. Even though he is capable of designing and producing mechanical modules, he will prefer standard, off the shelf modules over custom made. Mike notices that modules, both custom made and nuts and off the shelf, are typically made of many, typically standard, nuts and bolts.

The same productivity considerations applies in the work of Elia, the electrical engineer. To be productive, Elia prefers standard modules: keyboards, modems, and chip-sets to custom made modules, even though she is capable of making them herself. Electronic components: resistors, capacitors, transistors, etc., are Elia’s world equivalent of Mike’s nuts and bolts. If Elia needs, e.g., transistor and resistors for connecting standard modules, she prefer to select these from standards (JEDEC, 2017), e.g., an NPN polarity BJT transistor
(a 2N2222 or the like). Elia, just like Mike, notices that modules she buys, just as those she decides to make, tend also to use standard components.

The equivalent of modules in the software world are frameworks and libraries. Sveta, the software engineer member of the team, uses an external communication library, device driver, GUI framework and operating system, despite the alternatives of producing or subcontracting a custom made implementation.

The parallels drawn in the parable suggest that reuse occurs at different granularity levels: reuse at the “module” level is of elements of design that contain “components”. Modules are found in Mike, Elia and Sveta’s work. Transistors and resistors are Elia’s words for nuts and bolts: little things that are used in many ways and in different places in the system, either for connecting modules, or in the clockwork of the modules themselves.

But, what are nut and bolts in Sveta’s world?

1.2 Nano Patterns and Language

We suggest that nuts and bolts in the software world are the many little things that software engineers do repetitively: safe guarding against null, caching a returned value, delegating to an inner implementation, catching an exception only to ignore it, execute a command pending on some condition, a loop searching for an element in a given collection, etc.

We call these “little things that software engineers do” nano-patterns (nanos for short). Technically, a nano-pattern is a predicate over short code snippets that tries to match code written in response to an elementary programming mission e.g., “for each \( m \in M \) do ...”, or, “use \( x \) (but if \( x \) does not exist use null (or, some specific \( y \) instead)”. Not all little things are equal. Our interest lies with prevalent nanos: those that recur in substantial numbers across a variety of projects. Nano evaluateUnlessDefaultsTo is one nano that passed our prevalence threshold test. This nano can be written in pseudo-code as\(^1\)

\[
\text{[ Evaluate } \mid e \text{ unless } b \mid \text{ default } e' \text{ ]}
\]

The predicate behind evaluateUnlessDefaultsTo is designed to capture Java snippets such as

\[
(a \text{ instanceof } A) \ ? \ ((A)a).f() : \text{null}
\]

\(^1\)Here and henceforth we use these conventions: \( C \) for command, \( e \)-expression, \( p \)-predicate, \( M \)-multitude (which might be an array, stream, list, set, collection, etc.), \( b \) for Boolean expression, \( i \) for identifier, \( T \) for type, and, \( X \) for an eXception type.
\[
\text{pos > 0 ? pos - 1 : 0}
\]

\[
\text{if (b == 0)}
\]
\[
\text{return null;}
\]
\[
\text{return new Integer(a/b);}
\]

The contributed nano tracer makes the match, and may attach an appropriate code comment, transform the captured instance, or notify a listener of the event. Other prevalent nano-patterns are `letIn`, `firstSuchThat`, `pojo.Atomic.Constructor`, and `holdsOrReturn`.

Fig. 1 depicts our *taxonomized catalog*, or “language”\(^1\) of prevalent nanos.

Fig. 1: Taxonomy of the language of Java nanos

The 38 nanos of the language show as leaves of the tree in figure, except that some nanos have sub-nanos. There are too many nanos and sub-nanos to describe each individually, but Tables 1 and 12 below provide some hints. Example snippets from real code nanos of all is in App. B. Also, as in (1) nano names are often juxtaposition of the keywords in its pseudo code. For

\(^1\)The term “language”, endemic to the patterns literature, means roughly “useful catalog” or “domain-specific catalog”, see e.g., the conference series “Pattern Languages of Programs—PLoP” (PLoP, 2017) and other examples (Spinellis and Raptis, 2000; Zdun, 2007).
example, the pseudo code command \textbf{Let} \( i \leftarrow e \text{ in } C(v) \) (notations explained below), or in a slightly more general form,

\[
\text{Let } \cdot \leftarrow e \text{ in } C(\cdot)
\]  \hspace{1cm} (5)

(where “\(\cdot\)” denotes an anonymous free variable) is summarized in the nano name \texttt{letIn}. Similarly, the nano \texttt{firstSuchThat} is short for

\[
\text{First } \cdot \in M, \ p(\cdot)
\]  \hspace{1cm} (6)

Taxa are shown in the hierarchy of Fig. 1 as internal nodes, rendered with shade opacity increasing with granularity. Primary taxa correspond to syntactical elements. Terms \texttt{Composite} and \texttt{Atomic}, occurring as taxa in the secondary and tertiary levels, are interpreted in the context of recursive structure of language elements. For example, an \texttt{if} command is a \textit{composite} of one or two \textit{inner} commands, while a method invocation command is \textit{atomic} since it has no inner commands.

In a composite/atomic recursive hierarchy we say that constructs like “\texttt{if}”, “\texttt{synchronized}”, and, “\{\ldots\}” (curly brackets) are \textit{composite constructor} since they all are used by programmers to construct a composite command from inner command(s). An \texttt{Elaborator} taxon is of nanos that are similar to \texttt{synchronized} composite constructor in two respects: a \texttt{synchronized} composite command has a \(\text{(i)}\) single inner command, which it executes in a designated \(\text{(ii)}\) elaborated/protected execution environment. We use the term \textit{elaborator} for nanos which do essentially the same in their respective syntactical taxon.

For example, \texttt{defaultArguments} run a \(\text{(i)}\) single inner method in the \(\text{(ii)}\) elaborated execution environment that provides missing values for arguments. Nano \texttt{safeNavigation} (also known as the “\texttt{?}.” operator in, e.g., Swift and C\#) is an \texttt{Elaborator.Expression} nano—it executes an \(\text{(i)}\) inner expression in a \(\text{(ii)}\) \texttt{null} guarded execution environment.

The similarity between elaborators and the Java language constructs such as “\texttt{synchronized}” and the “\texttt{==}” operator is limited, since nanos try to identify a simple semantic action. And this action needs to be so simple that it is clutter, not essence, of design. Nano \texttt{executesUnless} is an elaborator, but is purposefully not as general as Java \texttt{if}. The predicate behind \texttt{executesUnless} represents the frequent need to conduct an atomic command on (say) a variable, but only if certain condition applies.

To summarize the notions of pattern, instance, and the various taxa, consider these key points related to the \texttt{safeNavigation} nano:

- The little (nano) thing that engineers do is \texttt{null}-guarding, e.g., trying
to code

\[
\text{State state} = \text{customer.getAddress().getState();}
\]

but writing instead,

\[
\text{State state} = \text{null;}
\text{if (customer!= null) }
\text{if (customer.getAddress() != null);}
\text{state} = \text{customer.getAddress().getState();}
\]

to guard for \text{customer} being \text{null} and \text{getAddress()} returning \text{null}.

- The nano is realized by a predicate on code which identifies the snippet (8) as (multiple) instance of \text{safeNavigation}. This identification is also called tracing.

  Interest lies with predicates that under-approximate patterns. In the case of this nano, the tracer is to demonstrate some refactorings which lead to a form using the “?.” operator, but it may err on the false negative side.

- Note that the nano is in the \text{Expression} taxon even though its trace might be a command.

- The nano is in the \text{Elaborator} taxon, since it executes its inner expression, in an environment which guards for \text{null}. The guarding here is similar to the protected execution environment offered by \text{try...catch}.\footnote{In fact, the protected execution environment that \text{safeNavigation} traces, can be Java encoded by the protected execution environment of \text{try...catch}.
}

The distinction between \text{Weak.Elaborator.Expression} and \text{Strong.Elaborator-.Expression} is described among other language details in Chap. 3.

### 1.3 Evaluation

Evaluation (see below Chap. 4) was across three corpora of 26 Java OSS projects each. Results indicate that the language prevails is substantial, e.g., about a quarter of all AST nodes are characterized by a nano, just as about half of iterative commands, and at least 40% of conditionals. These values are superior to prevalence of project specific idioms discovered automatically by Allamanis and Sutton (2014).

The language was initially refined on non-representative small training corpus (six projects of similar application domains; one of which somewhat
esoteric; two from the same development group). During this refinement, we did not examine its effect on any other code, nor did we check prevalence values in other projects.

The results were then submitted to empirical evaluation on a larger Baseline Corpus. This corpus was a readily available one, previously used in other (see e.g., Gil and Lalouche (2016)). The baseline corpus comprises three projects from the training corpus (by coincidence) along with 23 other projects. A second layer of evaluation was made on two corpora assembled for this research:

- The Java Corpus, comprising the 26 top most starred\(^1\), non Android, GitHub\(^2\), with the exception of projects in the baseline corpus.
- The Android Corpus, comprising the 26 top most starred, non Android, GitHub projects.

### 1.4 The Prevalence Threshold

Perhaps of its own interest, is the prevalence threshold, a criterion we use in the sake of making the selection of candidates more objective. The threshold is meant to be a robust and objective filter of domain specific nanos, and other coincidences. A nano-pattern passes the prevalence threshold if it passes the reuse threshold in, at least, \(\rho = 50\%\) of the projects in a corpus \(C\). The nano passes the reuse threshold of a project \(p\), if it is reused more times than the \(r\) most popular methods in \(p\). The \(r\) most popular (i.e, most reused) methods in the project are determined in a manner similar to that of selecting the publications that contribute to the \(h\)-index (Hirsch, 2005) of a scholar: A project has a method (class) reuse index \(r\) if it has \(r\)-methods (classes) which are reused \(r\) or more times. The prevalence threshold is parameterized by the corpus \(C\), and the fraction \(\rho\). Here we used \(\rho = 0.5\) and the baseline corpus as described in Chap. 4.

\(^1\)Time of writing
\(^2\)https://github.com
Outline

Outline. Context and applications are discussed in Chap. 2. The following Chap. 3 describes the language of nanos in greater detail. Chap. 4 then reports the empirical evidence for the language prevalence. Chap. 5 overviews the language’s evolution, from a list of candidates to a reasonably coherent language. The prevalence threshold and its ties with the $h$-index are the subject of Chap. 6. Related work and scientific background are in Chap. 7. Chap. 8 discusses threats to validity, and Chap. 10 concludes. Additionally, App. A broadens the scientific background further, and mentions other more related work. App. B gives code snippets exemplifying the nanos, and App. C provides essential reproducibility information. Finally, App. D provides more details on the Java like stenography we use occasionally in tables.
Chapter 2

Context and Applications

Source code is not random text. Just like natural language, it has regularities: words that occur more than others, idioms, clichés, recurring phrases, patterns, etc. (Gabel and Su, 2010; Tu et al., 2014; Hindle et al., 2012). A body of research in software engineering concentrates in identifying these regularities, and in the exploitation of these to benefit desirable ends such as productivity, identification of potential bugs, program comprehension, etc.

Nanos are similar to “idioms”, and other recurring, often nameless, pattern-like creatures found in code. Observe that the study of regularities of this sort tends to follow three main themes.

1. In the first, “opportunistic”, research theme, regularity is sought to serve a specific end. For example, work on code-completion (Hou and Pletcher, 2010) concentrates on those regularities that can make code completion more efficient. The history of research thus employing regularities is long, starting from early work on using these for purposes such as reverse engineering (Fiutem et al., 1996), software design (Koono et al., 2006), education (Upal and Padmanabhuni, 1999), program recognition (Wills, 1990), and automatic programming (Rich and Waters, 1988), continuing to contemporary work such as that of Sridhara et al. (2011) who observed that certain recurring code patterns represent “high level abstract algorithmic actions” which can then be converted into natural language description.

Regularities found by the first research theme often risk being ad-hoc and non-reproducible. Regularities found for the sake of (say) identifying algorithmic steps (or code completion) may be useful for better CASE tools, but the regularities found in one work are not necessarily encapsulated, named, and organized sufficiently to be applicable in similar domains.
The work of Wang et al. (2014) demonstrates: considering the problem of automatic segmentation of method bodies—these locations in the code in which insertion of new line increases code readability—the authors identified and named a number of recurring patterns, e.g., syntactically-same, data-flow chain, and extended-SWIFT. However, as of writing, the patterns found by Wang et al. did not find other applications.

2. In the second, “generative”, research theme, effort is invested in producing reusable regularity, i.e., identifying regularity that persists over a wide domain, or, more sophisticatedly, inventing techniques for discovering local regularity—the patterns and idioms peculiar to, e.g., an individual software artifact. Examples include, e.g., the work of Singer et al. (2010) on nano-patterns, and the seminal work of Hindle et al. (2012) on “naturalness”.

3. The third, “applicative”, research theme exploits the recurrences that were systematically found by generative studies for purposes similar to opportunistic study of regularity. Examples are plenty, e.g., the use of nano-patterns for fault detection (Deo and Williams, 2015) and in the context of security vulnerabilities (Sultana et al., 2016), the links found between micro-patterns and class names (Singer and Kirkham, 2008), the use of micro-patterns to predict class life time (Marion et al., 2007), bug introducing changes (Kim et al., 2006), and source code search engines (Bajracharya et al., 2006), not to mention the use of design patterns for program visualization (Schauer and Keller, 1998).

The present work follows the generative theme. Specifically, we offer a set of patterns that occur at the method and sub-method granularity level. We expect ensuing research to follow the applicative theme. Also, nanos border with language and library engineering, and the study suggests applications in these.

Nano-patterns are low granularity abstractions. For most high level software engineering purposes, they are implementation and language level details. Our results indicate that together these accumulate to clutter of non-negligible size. The language, the taxonomy behind it, the nano patterns in it, and the contributed automatic tool for tracing nano, provide means for managing and imposing order on this clutter. We see therefore applications of nanos in contexts such as program visualization, code comprehension and reverse engineering, clone and plagiarism detection, automatic documentation, etc.

Our work resembles the work is design patterns (Gamma et al., 1993), micro-patterns, and the two previous publications on nano patterns (Batarseh,
2010; Singer et al., 2010), in these respects:

- patterns being \textit{manually} generated, i.e., human expertise is employed for identifying commonalities,

- patterns are \textit{named}, i.e., the mental effort of identifying the patterns culminates in suggesting names such as \textsc{ChainOfCommand} design pattern, \textsc{Stateless} micro-pattern and \textsc{LocalReader} nano-pattern (of Singer et al.), and,

- patterns are Java \textit{universal} (rather than being artifact or project specific).

In contrast, regularities discovered by statistical algorithms (Gabel and Su, 2010; Hindle et al., 2012) identify anonymous recurrences which may be idiosyncratic to a specific input.

Also, the language is different from the previous work on nano-patterns (but similar to the work on design- and micro-patterns) in the following sense: Previous nano patterns languages concentrated in finding essentially \textit{orthogonal} properties of methods: A method can be both a \textsc{LocalReader} and \textsc{StraightLine} (even though the events are might be correlated). Design patterns, \(\mu\)-patterns and the language presented here focus in a specific and often \textit{peculiar incidence of properties} (see also below Sect. 7.3).

The literature is typically silent on the subjective process of discovery of patterns such as \textit{syntactically-same} or \textsc{Stateless}, i.e., what was it that made the authors choose these patterns rather than others. The community may find use of our description (Chap. 5) of the harvesting process we applied, and the rationale behind.
Chapter 3

The Nanos Language

We now discuss the language and the taxonomy (inspired by the writings of Watt (Watt)) of Fig. 1. The primary taxa of the hierarchy (for easy reference reproduced here, Fig. 2) divides the nanos by their syntactical role:

Three taxa, Field, Constructor, and, Method are the syntactical elements that make Java’s classes and types. Other nanos taxa are at the sub-method level—they occur in methods, constructors, and, initializers.

Sub-method nanos include Command (a synonym for an executable statement, i.e., excluding non-executable statements such as variable and type declaration), Expression, which syntactically occur in commands. The last
primary taxon, Phrase, groups two nanos at the sub-expression level: myName, the pattern of a method calling itself, or another identically named, overloaded version, of itself, and, myArguments, which is of a method to use a verbatim copy of its own argument list in calls to other methods.

The secondary and tertiary divisions in the taxonomy refer to the role of the nano in the composite–atomic relationship found in these syntactical elements, e.g., secondary taxon Atomic.Command gathers atomic command nanos. This is to say, nanos which look like an atomic command. Nanos that capture code whose which has the structure of inner command(s) in a compound, are gathered in secondary taxon Composite.Command. These include:

1. Nano whenHoldsOn capturing a recurring case in which a programmer applies a predicate to series of values, taking action based on the first value which satisfies the predicate. Writing this nano in pseudo code, shows that it is a compound command assembled from inner commands:

\[
\text{When } p(x) \\
of x, y, z: C_1 \\
of w: C_2 \\
otherwise C_3
\]


3. Tertiary taxon Iterative.Composite.Command of nanos that capture the little recurring task of making a “simple” iteration over a collection, a sequence, an array, etc. In “simple” we mean that predicates behind the nanos restrict iteration body to be an atomic command, in form suitable e.g., for adaption to Java 8 streams API.

The Expression sub-hierarchy includes:

- taxon Multitude.Expression with the nanos that represent recurring rudimentary operators (of languages other than Java) over non-scalar values.

- taxon Elaborator.Expression, which includes rudimentary (non-Java) operators guarding from exceptional situations in the computation of their operand. The distinction between Weak.Elaborator.Expression and Strong.Elaborator.Expression is based on the permitted correction action. Strong elaborators throw an exception, weak ones may not do so.
The distinction between elaborators and atomic nanos carries also to methods and constructors. All nanos in Elaborator.Method are methods whose body is essentially delegation to one other method. Each of the particular nanos in the taxon makes its own peculiar demands on this method call. Similarly, all nanos in Atomic.Method taxon are methods whose very simple body does not include any method calls, and is limited. Each of the nanos in the taxon makes its own strict demands on the simplicity of this call-less body. The sub-hierarchy of Constructor is smaller, but similar, including Atomic.Constructor and Elaborator.Constructor.

The metaphor is extended also to fields, distinguishing between Atomic.Field which include no field-access, and Elaborator.Field nanos which elaborate this access. As indicated by the dashed line in Fig. 1 and Fig. 2, taxon Elaborator.Field specializes Atomic.Method, i.e., methods which make no calls to other methods. In the case of Elaborator.Field, the restriction placed on this method are to either retrieve the field’s value (getter), set this field’s value (setter), or set the field’s value and then return this; (fluent-setter).

Table 1 provides more details on each of the nanos. This table is complemented by Table 12 providing below similar information on sub-nanos. We do not discuss sub-nanos further than this.

Column (*) a in the above traces back nanos to their origin, the nano-candidate(s) from which they evolved. The language evolution can be learned by comparing nanos with their legacy candidates (see below Table 4). The next column, (*) b, shows each nano’s prevalence in the baseline corpus. The score and the filtering process are described below in Chap. 5 and Chap. 6.

The information in the subsequent columns highlight places in which the study of nanos borders with language design and library design:

- The name of the nano is often derived from its encoding in pseudo code language, and we use pseudo code for describing the intent of some nanos.

- Some of the nanos, e.g., questionQuestion, occur as operators in other programming languages.

- Even though nanos are not JSRs, it may be possible to extend the language to support shorter encoding of some nanos. In describing some nanos, we employed a “stenographic Java” notation, which should be self explanatory, but interested readers may consult App. D for more information.

- Column (*) c in Table 1 marks those nanos that can be encoded as Java
8 library service, e.g., the occurrence of `defaultsTo` in the code snippet

\[
\text{state} = \begin{cases} \text{customer.getAddress().getState() == null ? california :} \\
\text{customer.getAddress().getState();} 
\end{cases}
\]

(2)

can be replaced by

\[
\text{state} = \text{de.fault(customer.getAddress().getState()).to(california);} 
\]

(3)

which is valid Java, assuming these definitions of `de`, `faults(·)` and `to(·),`

\[
\text{public interface de {} \\
\text{interface To<T> { T to(T t); } \\
\text{static <T> To<T> fault(T to) { return t -> t != null ? t : to; } }
\}
\]

(4)

Indeed, nanos border with language and library engineering. One potential application is a nano-library to encode nanos, with fluent API call chain and λ-expressions. Examining Table 1 we see this is possible for about half of the nanos, all located in the left half of the table. (The current Java 8 stream library covers almost all nanos in the lower part of the table.)

Nanos show a couple of themes of little things that engage programmers.

- **Defaulting and guarding:** Nanos in the first block of the table are of making small corrections of small errors and exceptional values, substituting a missing value with some default, terminating the current method and returning a parameter is null, or when a pre-condition is not met, etc. This theme shows in `safeNavigation`, but also in many other nanos.

- **Simple operations on multitudes:** as shown in the second table block.

- **Pipes and sinks:** as shown in the first half of Table 1, there are a number of nanos in which methods and constructors are used for adjustment, adding parameters, delegating to a receiver, etc. In this half, we also see cases of “sinks”, methods that call no other methods and whose body is trivial, standard constructor body, etc.

The themes may find their application in evolution of new programming languages and the extension of existing ones. Language designers should consider substituting these “common nuts and bolts” with language builtin, with dedicated syntax and less-traditional operators.
In our parable, we can say that language mechanisms standardize nuts and bolts. Software libraries do the same standardization, but offering a different tradeoff: language syntax is more powerful than libraries, but, libraries are easier to design and change.

A certain balance between flexibility and expressiveness is found in macro processing (Lee et al., 2012) and syntax sugaring (Erdweg et al., 2011). These techniques might also be used towards the end of standardization of nuts and bolts. In particular, macros and sugaring may deal better with the smaller theme of reference to self (the two nanos in the Phrase) are impossible to standardize with library, and probably too drastic.

Conversely, the themes are also material in domains such as visualization and clone elimination, and, code comprehension, in which the clutter of nuts and bolts should be hidden, not eliminated. The effectiveness of automatic document generator should be better if it can classify more portions of the code as boilerplate.
<table>
<thead>
<tr>
<th>Name</th>
<th>Intent</th>
<th>Expression and Command Elaborators</th>
<th>Operations and Operators on Multitudes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
<td>Execute C unless b</td>
<td>Execute C when b</td>
<td>Aggregate M with A(. ·)</td>
</tr>
<tr>
<td>C₂</td>
<td>executeUnless</td>
<td>When p() of x, y, z: C₁ of w: C₂ otherwise C₃</td>
<td>Select i ∈ M s.t. p(i) {x ∈ M</td>
</tr>
<tr>
<td>C₃</td>
<td>whenHoldsOn</td>
<td>If (F₁ or F₂ or F₃ or ...) do ...</td>
<td>For i ∈ M s.t. p(i) {x ∈ M</td>
</tr>
<tr>
<td>E₂</td>
<td>defaultsTo</td>
<td>e₁ default e₂</td>
<td>M/filter(...).reduce(...)</td>
</tr>
<tr>
<td>E₃</td>
<td>questionQuestion</td>
<td>e₁ default e₂ else e₃</td>
<td>M/filter(...).collect(...)</td>
</tr>
<tr>
<td>E₄</td>
<td>safeNavigation</td>
<td>eǐ.f() eǐ.m()</td>
<td>M/filter(...).forEach(...)</td>
</tr>
<tr>
<td>E₅</td>
<td>evaluateUnlessDefaultsTo</td>
<td>v = [Evaluate] e unless [default e']</td>
<td>First(M/p)</td>
</tr>
<tr>
<td>F₁</td>
<td>throwOnFalse</td>
<td>Throw X when b</td>
<td></td>
</tr>
<tr>
<td>F₂</td>
<td>throwOnNull</td>
<td>Throw X when e nulls</td>
<td></td>
</tr>
<tr>
<td>F₄</td>
<td>notNullRequired</td>
<td>Return default when a nulls</td>
<td></td>
</tr>
<tr>
<td>F₅</td>
<td>notNullAssumed</td>
<td>Return default when e nulls</td>
<td></td>
</tr>
<tr>
<td>F₆</td>
<td>holdsOrReturn</td>
<td>Return default when b</td>
<td></td>
</tr>
<tr>
<td>F₇</td>
<td>ignoringExceptions</td>
<td>{ ... } ignoring X₁, ..., Xₙ;</td>
<td></td>
</tr>
<tr>
<td>M₄</td>
<td>letInNext</td>
<td>let v ← e in C;</td>
<td></td>
</tr>
</tbody>
</table>

(*): a, b, c

<table>
<thead>
<tr>
<th>(*)</th>
<th>Name</th>
<th>Intent</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Expression and Command Elaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
</tr>
<tr>
<td>C₂</td>
</tr>
<tr>
<td>C₃</td>
</tr>
<tr>
<td>E₂</td>
</tr>
<tr>
<td>E₃</td>
</tr>
<tr>
<td>E₄</td>
</tr>
<tr>
<td>E₅</td>
</tr>
<tr>
<td>F₁</td>
</tr>
<tr>
<td>F₂</td>
</tr>
<tr>
<td>F₄</td>
</tr>
<tr>
<td>F₅</td>
</tr>
<tr>
<td>F₆</td>
</tr>
<tr>
<td>F₇</td>
</tr>
<tr>
<td>M₄</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Phrase</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>Field</td>
</tr>
</tbody>
</table>
| 22 | A_2 | 77% | const | $\pi \equiv 3.141593$  
|     |     |     |     | const $\Pi = 3.14159$; |
| 23 | K_3 | 88% | setter | $\text{protected } \leftarrow \text{private } T i = e;$ |
| 24 | K_4 | 100% | getter | $\text{public } \rightarrow \text{protected } T i = e;$ |
| 25 | K_5 | 54% | fluentSetter | $\text{public fluent } \leftarrow T i = e;$ |
| 26 | K_6 | 58% | vanillaCollection | $\text{List}(T) = \text{new ArrayList}()$;  
|     |     |     |     | $\text{Map}(K,T) = \text{new HashMap}()$;  
|     |     |     |     | $\text{Set}(T) = \text{new HashSet}()$; |
| Atomic Method |   |   |   |   |
| 27 | J_1 | 85% | constantValue | $T f(...) = c$;  
|     |     |     |     | return $c$; arguments (if any) are ignored |
| 28 | J_4 | 73% | thrower | $T f(...) = \text{throws } X$;  
|     |     |     |     | $T f(...) = \text{default}$; |
| 29 | J_5 | 100% | defaultValue | $\text{return } c$; depending on $T$ returns $\text{null, 0, false, or nothing (if } T \text{ is void); arguments if any are ignored}$ |
| 30 | L_4 | 100% | examiner | $T f(...) = c$; |
| 31 | M_5 | 100% | letIn | $T f(...) = \text{let } v \leftarrow e \text{ in } C$; |
| Elaborator Method (Single Call) |   |   |   |   |
| 32 | L_1 | 96% | factory | $T f(A_1, \ldots, A_n) \rightarrow \text{new } [A'_1, \ldots, A'_n]$; |
| 33 | L_2 | 85% | defaultArguments | $T f(D_0, A_1, D_1, \ldots, D_{n-1}, A_n, D_{n+1})$; |
| 34 | L_3 | 69% | adjuster | $T f(A'_1, \ldots, A'_n) \rightarrow g$; |
| 35 | L_4, L_5 | 100% | delegator | $T f(A_1, \ldots, A_n) \rightarrow g(A_1, \ldots, A'_n)$;  
|     |     |     |     | $T f(A_1, \ldots, A_n) \rightarrow a_i : g \{A'_1, \ldots, A'_n\}$;  
|     |     |     |     | $T f(A_1, \ldots, A_n) \rightarrow \text{super } \{A'_1, \ldots, A'_n\}$; |
| Constructor |   |   |   |   |
| 36 | N_1 | 100% | pojo | $T(A_1, \ldots, A_n) \rightarrow \text{default}$; |
| 37 | N_2 | 88% | super | $T(A_1, \ldots, A_n) \rightarrow \text{super } \{A'_1, \ldots, A'_n\}$; |
| 38 | N_3 | 81% | this | $T(A_1, \ldots, A_n) \rightarrow \text{this } \{A'_1, \ldots, A'_n\}$; |

- Candidate(s) from which this nano was created
- Prevalence score in the baseline corpus
- Can be emulated by lazy-evaluation, using lambda functions
- e.g., pseudo-code, Java 8 streams, suggestion for extending plain Java syntax, etc.
Chapter 4

Language Prevalence

Table 2 shows the prevalence (in percent point) of the nano in three corpora. *Prevalence* is defined as the fraction of syntactical elements, expressions, statements, methods, etc., that are matched by one or more nanos.

A quick takeaway of Table 2 is the language’s prevalence is substantial: nanos in the language prevail over about third of all commands, half of iterative commands, 80% of conditional commands that can be converted to conditional expressions, and 40% of the remaining conditional commands. About 55% of all methods are characterized completely by the language, while in about a third of all other methods one or more nano is found.

We note that prevalence of our subjectively selected language rivals that of patterns discovered by an automatic learning process (Allamanis and Sutton, 2014). (This comparison is done in terms of AST node coverage. No data was available for the other measures we use.)

The first corpus in table is the *baseline* corpus. It is distinguished from the two corpora in that is took part in the language evolution, and can therefore be thought of as training corpus. The baseline corpus constitutes 26 popular projects (Gil and Lalouche, 2016) selected from of *GitHub’s TrendingWikiBookJavaIdiom repositories*¹ augmented by *GitHub Java Corpus*² list due to Allamanis and Sutton (2013). Reproducibility and other information on projects constituting this corpus are listed in Table 8. Overall, the corpus represents the work of circa two thousand programmers and at least a century of overall development time.

The two other corpora in Table 2 can be thought of as testing corpora since language evolution and pattern harvesting had no access to these.

- The second corpus in the prevalence table (Table 2), consists of the 26

1<https://github.com/trending?l=java&since=monthly>
2<http://groups.inf.ed.ac.uk/cup/javaGithub/>
Table 2: Aggregating statistics of prevalence (in percent point) of the nanos language in three corpora of 26 OSS projects each.

<table>
<thead>
<tr>
<th></th>
<th>Commands</th>
<th>Expressions</th>
<th>Nodes</th>
<th>Methods</th>
<th>Control Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
</tr>
<tr>
<td>Baseline Corpus</td>
<td>33</td>
<td>34</td>
<td>32</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>Java Corpus</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>Android Corpus</td>
<td>30</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
</tbody>
</table>

- The third corpus in the prevalence table (Table 2), consists of the 26 top most starred Java Android projects in GitHub, excluding projects included in the baseline corpus. Reproducibility information can be found in Table 9.

The two test corpora were sampled at the time of writing. In contrast, the last update in the training corpus was 2014-8-21 (in project Hadoop-common).

Examining the table, and comparing the respective prevalence values in the three corpora, we can see that the numbers tend to be lower in the test corpora then the training corpus. Still, prevalence is high, regardless of corpus, measure of prevalence, and the summarizing statistic, and the differences, when present, are relatively small.

We also run a battery of Wilcoxon signed-rank tests for statistical significance. The results however were unequivocal: The null hypothesis is that the prevalence values (for a measure such as ”conditional expression”) found in two corpora, one of which is training and the other test, are drawn from the same distribution. As it turns out, the null hypothesis was not rejected for all prevalence measures, and when it was, the rejection sometime indicated that the testing corpora were better than the training corpus, or that the differences between the two testing corpora were as significant as the difference
with the training corpus.
Chapter 5

Language Evolution and Implementation

This section reviews the language evolution. To our knowledge, this is the first description in the literature of the subjective process of pattern harvesting. In the first phase of language evolution, an extensive number of initial “candidates” underwent subjective evaluation and refinement, in which we tried to identify candidates that are

- *traceable* (within the limitation of our nano tracer),
- likely to be *prevalent*,
- *purposeful*, e.g., by writing as familiar pseudo-code, or a description of a small programming activity.

Candidates surviving this *subjective* scrutiny phase, went in the second phase to an *objective* prevalence threshold against submitted to further evaluation in the *baseline* corpus; the language was then taxonomized from candidates that either passed this test, or could have been rephrased to pass it.

Recall that nanos are predicates on code designed to spot a small semantic action of low granularity. They are realized by our nano tracer, a lightweight Java library wrapping Eclipse’s JDT, for matching and substitution on the abstract syntax tree. Upon identifying an instance of nano in the code, the tracer can e.g., inject a code comment representing the nano trace.

**Fig. 3**, showing our predicate implementation of `defaultsTo`, demonstrates the capabilities (and limitations) of the tracer employed.
public final class DefaultsTo
    extends NanoPattern<ConditionalExpression> {
    public DefaultsTo() {
        super.requirements.andAlso(
            findReplace("$X1 != null ? $X1 : $X2", "de.fault($X1).to($X2)"
                ).or("$X1 == null ? $X2 : $X1", "de.fault($X1).to($X2)"
                ).or("null != $X1 ? $X1 : $X2", "de.fault($X1).to($X2)"
                ).or("null == $X1 ? $X2 : $X1", "de.fault($X1).to($X2)"
                )
        );
    }

    @Override
    public boolean applicable(final ConditionalExpression x) {
        return requirements.applicable(x);
    }
}

Fig. 3: Class for tracing the defaultsTo in Java code

The string literals in the figure show a small DSL for structural match-and-replace. In fact, the code in the figure replaces code such as (2) with (3) (assuming the library (4)). The tracer also features local name binding and rudimentary type inference, but not data flow nor control flow analysis.

Instead of these advanced analyses, prior to actual matching, the tracer makes a number of semantic preserving small code transformations, designed to minimize style variations and to bring the code to a shorter, more canonical, form. These transformations include inlining of temporary variables when possible, simplifying boolean expressions, remove redundant parenthesis, bringing loops to canonical form, converting if...else commands to conditional expressions, etc.

5.1 Phase 0: Candidate Collection

We collected candidates following these three steps repeatedly:

1. Implement the predicates that approximate a new pattern idea on the tracer,

2. examine the code snippets to refine the predicates, and,

3. reflect whether to refine, defunct, and, reorganize existing patterns and to raise new patterns ideas.

These steps were carried out against a (biased) corpus of 6 OSS projects (depicted in Table 3). Overall, this “preliminary” corpus is not small—spanning circa 300,000 lines of code. The manual collection of candidates was done by manual inspection of code samples drawn from the corpus. But, this sampling was in no particular order nor attention to intra-corpus bias (i.e., sampling not giving equal weight to projects in the corpus).
Table 3: Corpus of 6 OSS Java projects used for preliminary testing: identification, reproducibility information and size metrics.

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>Date</th>
<th>Hash</th>
<th>Lines (K)</th>
<th>P (^a)(K)</th>
<th>T (^b)(K)</th>
<th>M (^c)(K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmosphere</td>
<td>2.1.1</td>
<td>2014-04-12</td>
<td>521350f</td>
<td>31</td>
<td>20</td>
<td>0.4</td>
<td>3</td>
</tr>
<tr>
<td>Commons-bcel</td>
<td>BCEL_6_0</td>
<td>2016-07-14</td>
<td>7b9e151</td>
<td>47</td>
<td>12</td>
<td>0.5</td>
<td>4</td>
</tr>
<tr>
<td>Commons-lang</td>
<td>LANG_3_5</td>
<td>2016-10-20</td>
<td>a9a3cef</td>
<td>46</td>
<td>15</td>
<td>0.2</td>
<td>3</td>
</tr>
<tr>
<td>Guava</td>
<td>v20.0</td>
<td>2016-10-28</td>
<td>32130d5</td>
<td>113</td>
<td>32</td>
<td>2.0</td>
<td>15</td>
</tr>
<tr>
<td>JUnit</td>
<td>r4.12</td>
<td>2014-12-04</td>
<td>e87c6fc</td>
<td>11</td>
<td>44</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>Spartanizer</td>
<td>v2.8.0-beta</td>
<td>2017-01-23</td>
<td>10b19af</td>
<td>44</td>
<td>42</td>
<td>0.5</td>
<td>6</td>
</tr>
</tbody>
</table>

Total: 6 projects

\(^a\) number of packages
\(^b\) number of types (classes, interfaces, enums), in thousands
\(^c\) number of methods, in thousands

5.2 Phase I: Candidate Pruning

Table 4 depicts the 82 candidates thus collected. Every nano in the language of Fig. 1 evolved from one or more of these candidates, but not all surviving candidates became nanos.

Table 4 is not intended to be exhaustive—we focus on the more promising patterns. Some obvious absences are some higher order functions (e.g., currying, composition, and folding), set operations (intersection, union, complement, etc.), zipping and streams of pairs, element swapping, and many more. These may be pursued in continuing research of identifying higher level operations of one language in the code of another.

Candidates selection and categorization was carried out by brainstorming, trying to draw inspiration from experience in other languages and programming environments (and to a certain extent, from the nuts and bolts parable). The intent column in the table records the source of inspiration, including first order predicate logic, programming for the vi editor, experience in Eclipse and programming languages, from ancient Pascal to contemporary Swift.

The rationale behind seeking this inspiration is that designers of languages and environments included in their design abstractions and mechanisms that help programmers in their recurring nano activities: Candidate C\(_1\) in the table represents the conjecture that the \texttt{while...else} control of Python would be found as a nano of Java. Search for nanos of classical Lisp processing, was through candidates I\(_1\) and I\(_3\). We were unable to trace these specific Python and Lisp nanos in our corpus. But, other nanos such as \texttt{const}, \texttt{forEachdo} owe their roots to Pascal.

There are too many candidates in the table to describe each individually. The division into 14 taxa can guide a quick survey. As typically in the case of intuitive free form, opportunistic search for recurrence, some of the candidates
Cast reference to integer:

Comma separated list of names of parameters

Find the min/max of two numerical values

An

Constant definition—a

C

Java

Method invoking an overloaded (different sanity) version of itself

A. Recurring phrase

C

Pascal

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

Cast boolean to integer:

☡

ML

e

Assume, inside a method, that an expression is

Swift

☡

☡

VIM

Scala/SQL

☡

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

☡

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

☡

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

☡

VIM

Scala/SQL

��

 Kapoor

 Castillo

Java

C

Pascal

 Kapoor

Swift

☡

نصوص مختارة
are similar, and division into taxa is not absolute. It is also interesting to compare the taxa in the candidates table with the form they took in the taxonomy (see Fig. 1 and Table 1 above).

Taxon A is about phrases and language features that do not correspond to the familiar syntactical constructs and mechanisms of the subsequent taxa. Taxon B represents the automatic conversions between scalar types, typical of the C language. Taxa C, D, E represent not so-usual control flow (and in particular–iteration) and scalar operators such as Elvis. Noticing that many of these operators tend to be used for managing exceptional situations or null value, taxon F was made the target of a broader search for candidates that deal with these situations. More operators and operations on sequences and other non-scalars are taxa G, H and I.

A number of taxa represent our experience in examining the code of short methods.

- Taxon J represents the many short methods which are sink of the control flow graph (stub) and whose body is a simple expression of a peculiar kind.
- Boilerplate methods (L) are those whose body is a single call to another method,
- Taxon N represents nanos of method constructors.

Taxon K is dedicated to nanos on fields; these nanos are often methods as well. Finally, taxon M is dedicated to nanos which combine commands and expressions, and are not as small as phrases.

5.3 Phase II: Candidates to Nanos

Table 4’s legend summarizes the fate of the candidates: 9 candidates were abandoned at initial stage, when the small number of instances detected in the preliminary study corpus did not justify (in our mind) further study. Candidates are described in the table by their “stenographic Java” comment syntax.

The remaining candidates went then to prevalence testing on the baseline corpus (Table 8). A total 38 candidates were included in the language, and 10 candidates were merged into other nanos. The tested, yet rejected candidates are enumerated in Table 5.

The first block of the table enumerates 10 candidates whose prevalence was below \( \rho = 50\% \) in the baseline corpus. The two other blocks of the table are of candidates which are in fact missed opportunity for a “simple”
Table 5: Candidates phrased as nanos, but rejected later

<table>
<thead>
<tr>
<th>(*)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rare Candidates</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F₀₀</td>
<td>0%</td>
<td>rethrowExceptionAs { ... } rethrowing [ X₁ \</td>
</tr>
<tr>
<td>F₇₀</td>
<td>6%</td>
<td>returnOnException { ... } returningOn [ X₁ \</td>
</tr>
<tr>
<td>J₁₀</td>
<td>19%</td>
<td>returnThis \ T f() = this;</td>
</tr>
<tr>
<td>J₁₂</td>
<td>19%</td>
<td>returnParameter \ T f(a, ...a) = a;</td>
</tr>
<tr>
<td>J₆₇</td>
<td>15%</td>
<td>caster \ T₁ f(T₂) = caster;</td>
</tr>
<tr>
<td>K₄₁</td>
<td>4%</td>
<td>returnPrevious \ old v = e; (assign e to v, but return v’s old value)</td>
</tr>
<tr>
<td>K₅₂</td>
<td>15%</td>
<td>lazy \ T f = e lazily;</td>
</tr>
<tr>
<td>L₇₃</td>
<td>12%</td>
<td>mapper \ T₁ f(T₂) = mapWith g(·);</td>
</tr>
<tr>
<td>M₂₄</td>
<td>0%</td>
<td>exhaust \ exhaust e</td>
</tr>
<tr>
<td><strong>Rare Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₄₅</td>
<td>0%</td>
<td>min/max \ min(·,·)/max(·,·)</td>
</tr>
<tr>
<td>B₂₃</td>
<td>23%</td>
<td>asBit \ as.bit(e)</td>
</tr>
<tr>
<td>F₃₄</td>
<td>0%</td>
<td>putIfAbsent \ M.putIfAbsent(k, v)</td>
</tr>
<tr>
<td>G₄₅</td>
<td>35%</td>
<td>isSingleton \ is.singleton(M)</td>
</tr>
<tr>
<td>H₁₉</td>
<td>0%</td>
<td>count \ M.size()</td>
</tr>
<tr>
<td>H₂₉</td>
<td>0%</td>
<td>copy \ v = copy.of(M)</td>
</tr>
<tr>
<td>I₈₂</td>
<td>4%</td>
<td>last \ v = last(M)</td>
</tr>
<tr>
<td><strong>Prevalent Function</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₈₁</td>
<td>77%</td>
<td>first \ v = first(M)</td>
</tr>
</tbody>
</table>

*a* Candidate(s) from which this nano was created

*b* Prevalence score in the baseline corpus

method call, i.e., a method not relying on inner classes, function arguments, nor function return value.

As shown in the table, prevalence of all but one of these method nanos was below the set threshold. The only exception is the “first” nano candidate. It was rejected in the sake of language uniformity: All nanos in the language are not missed opportunities of simple method abstraction.
Chapter 6

The Prevalence Threshold

Not all patterns can be encoded in Java, and nano tracing is useful, even without modifying the code at all (for purposes such as visualization, code summary and documentation, fault location, etc.) Still, this section draws some parallels between nanos encodable as lambda expressions, to describe a rationale for our use of the r-index.

Consider nano `defaultsTo`, and its implementation as fluent API call chain `default(.).to( )` via the four-line definition (4). Indeed, the nano can be abstracted and captured as fluent chain. But, the introduction of even the short definition (4), just like any other library code, has its maintenance costs. Is the capture worth the effort? In the language of the starting parable, is this “reuse of a standard bolt” worth the trouble of standardization?

The question is similar to that of eliminating code duplicates. Upon identification of duplication of a certain cardinality, an engineer should consider abstracting over the duplication, representing it in (say) a method, and then refactor the sites of duplication to invoke this method. This consideration must however take into account that reuse has its maintenance and other costs (Tomer et al., 2004).

A conservative and greedy project management policy is adopting a personnel management guideline, “to avoid degrade, recruit above your grade”. To understand why, assume that reuse is a valuable end, measured at the project level by some agreed upon “reuse grade”. The incurred cost of a reuse opportunity, is to be weighed against this grade. The recruit above your grade policy offers two nice properties:

- Globally, each new addition is justified by specific increase to the overall grade. If no reused components are ever erased, grade will increase indefinitely.

- A reuse opportunity that does not increase the grade can be rejected
without considering its cost.

The consequence is that candidates for abstraction (of nanos and duplicate elimination) that do not increase the supposed reuse grade, could be rejected immediately. The actual grade-increasing of remaining abstraction and reuse candidates is to compared with their (typically unknown) cost.

In using the grade increase as the sole criteria for evaluating nanos, we effectively assume that the cost of using captured nanos is relatively small (but still positive). In truth, this reuse cost (e.g., learning a new language feature or library of tricks) is amortized across the many projects in which a nano is used.

It is tempting to define the grade of the project as the mean reuse level of its components. However, the long tail distribution of reuse level components in the library to grade a project. However, will prevent this grade from being effective. The average number of locations in which a method is called is often less than two. The recruit above your grade would mean that nanos occurring just twice should be not be rejected. A similar issue lines the median and other simple statistics.

The h-index proved robust in grading scientific proliferation, which due to the long tail distribution of citations, resist statistics such as mean and median. The r-index adopts the idea as a robust grade of the reuse level of a project.

The famous definition of Hirsch (2005) can be rephrased as

the $h$-index of a scholar $s$ is $k$ if $k$ is the maximal value for which satisfies the statement:

Scholar $s$ has $k$ publications which are all cited $k$ or more times.

\begin{equation}
\text{Scholar } s \text{ has } k \text{ publications which are all cited } k \text{ or more times.}
\end{equation}

By this definition, if the $h$ index is $k$, then $s$ gathered at least $k^2$ citations. Similarly, the r-index of a software project $p$ is $k$ if $k$ is the maximal value $k$ which satisfies the statement:

Project $p$ has $k$ distinct cases of code reuse, each of which recurring $k$ or more times.

\begin{equation}
\text{Project } p \text{ has } k \text{ distinct cases of code reuse, each of which recurring } k \text{ or more times.}
\end{equation}

The above definition may be applies to reuse of classes, methods, and any other modular structure. We concentrate in reuse of methods. A use instance of a method is a code location in which it is called. With the exception of the first use instance, all other use instances of a method, are reuse instances.
Table 6: Summary statistics of r-index values in the baseline corpus (Table 11)

<table>
<thead>
<tr>
<th>Statistics</th>
<th>r</th>
<th>r_i</th>
<th>r_x</th>
<th>r_i - r_x</th>
<th>Δr</th>
<th>Δr_i</th>
<th>Δr_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>46.7</td>
<td>40.7</td>
<td>24</td>
<td>16.7</td>
<td>20.2</td>
<td>18.2</td>
<td>7.9</td>
</tr>
<tr>
<td>SD</td>
<td>25.5</td>
<td>22.8</td>
<td>12.1</td>
<td>13.3</td>
<td>18.8</td>
<td>16.8</td>
<td>9.2</td>
</tr>
<tr>
<td>Median</td>
<td>41.5</td>
<td>37</td>
<td>21.5</td>
<td>13.5</td>
<td>15</td>
<td>13.5</td>
<td>6</td>
</tr>
<tr>
<td>Quartiles</td>
<td>26.5–60.8</td>
<td>23–51</td>
<td>14.2–29</td>
<td>5.8–24.2</td>
<td>7.8–19.5</td>
<td>8–19.2</td>
<td>2.2–9.5</td>
</tr>
</tbody>
</table>

Thus, if the r index is k, then p gathered at least k² distinct cases of code reuse. A nano used n times in p passes the reuse threshold of p if n ≥ k, where k is the value of the r-index of p.

A nano passes the reuse threshold of a certain project p, if it is reused more times than the r most popular methods in p, where, the r most popular methods in project p are defined with respect to the reuse index (for short, the “r-index”) of p, i.e., (2).

Suppose for example that a certain nano was detected 99 different times in project wildfly. To determine whether this nano passed the reuse threshold of this project, 99 should be compared to the r-index of wildfly. Consulting Table 11 int the appendix we see that the r-index of wildfly is 106, and therefore the nano did not pass the reuse threshold of wildfly. The number of occurrences of the nano in each project has to be compared with the r-index of this project. To be prevalent, the nano has to pass the reuse threshold of 13 projects or more.

Table 6 gives summary statistics of the distribution of of r-index values in the baseline corpus. The makes the distinction between r_i, the internal r-index in which only methods defined in project p are counted, and r_x, the external r-index, in which only methods defined externally to p are counted (i.e. External libraries, API, etc.). When no subscripts are attached to r, the index does not discriminate between reused methods based on their definition’s location.

The last three columns of Table 6 show the increase in values of the three indexes over the monitored period in the evolution of the baseline corpus. As expected, all these grades increase as projects evolve.

Notice that the r_i grade is typically greater than the r_x grade. Overlooking the slight bias of a project overriding methods defined in external code, this difference indicates that projects tend to reuse their own code rather than external code.

A priori, nanos which are not indented to be project specific, are to be compared with the lower r_x grade, since they are not-project specific. We used however the higher r threshold, while giving partial score to nanos that fail this threshold, but still met the r_i or r_x thresholds.
Ending this section, we note that our technique for finding the core of reuse (most popular methods) in an artifact may be effective in other cases where the underlying distribution is long tail (Egghe and Rousseau, 2006). Such distributions are abundant in the many software engineering contexts (Cohen and Gil, 2000; Concas et al., 2007, 2011; Turnu et al., 2011).

For example, to gather the set of main contributors to a software artifact, instead of considering the smallest set of authors whose accumulative contribution is 80% or some other arbitrary threshold, we might use the author index (a-index) of the artifact (defined as the largest a for which there are a authors of, at least, a classes) and only consider the authors whose contribution is no smaller than the a-index. It might be worthwhile to correlate projects’ success (measured, e.g., by popularity) with its r-index and a-index, and see whether these are better predictors than plain counts of authorship and reuse.

Similarly, in evaluating the eco-system (Decan et al., 2017) of package repositories, we can define the reuse index (r-index) of the repository based on the number of times a package is used by other packages in the repository. The reuse-core of the eco system is defined as the set of packages that contribute to the r-index of the repository.
Chapter 7

Related Work

7.1 Design Patterns and Micro Patterns

A “design pattern” is a specific composition of object oriented components (classes, methods, inheritance) and captures a recurring solution for a common design problem (Gamma et al., 1995). Interest, both practical and research oriented, in design patterns thrives ever since the initial gang-of-four’s announce of their discovery (Gamma, 1997). A “micro pattern” (Gil and Maman, 2005) is a predicate on OO types (classes) that capture a peculiar restrictive use of OO features, e.g., no fields, or just one field, no methods, strict restriction on constructors, inheritance, etc. Compare µ-patterns to design patterns, noting that

- µ-patterns are prevalent. Empirical evidence is that around 75% of all classes in Java projects can be categorized as one or more of 27 or so µ-patterns. Demands on design patterns are more modest. To be worthy, they are to be encountered in a handful of projects.

- design patterns are limited in not being traceable, i.e., they are not automatically recognizable. See, for example, the classical van Emde Boas lament (Van Emde Boas, 1997), and the many attempts to formalize these (Eden et al., 1999, 1996; Eden, 2001, 2002, 1998), and to automatically detect these in source code (Ferenc et al., 2005; Guéhéneuc et al., 2010; Zanoni et al., 2015; Guéhéneuc et al., 2004; Vokác, 2006; Wang et al., 2012). Micro-patterns, in contrast, are traceable.

- design patterns have an attached purpose, a worthy design pattern is one that addresses a specific problem. No such purpose is attached to µ-patterns: they do not answer to a specific design problem, but capture a programming technique frequently used by software engineers. It was
argued (Gil and Maman, 2005) that prevalence is an indication that a \( \mu \)-pattern is \textit{purposeful}, i.e., that there was one (or more) design problem that drove engineers to abide by the pattern’s restrictive predicate. Alas, this indication is only a clue to the folk-lore behind these presumed purposes.

Judging nanos (Fig. 1) by these three, we have that they are \textit{prevalent} (Chap. 4) and \textit{traceable} (by manner of construction of a nano tracer (Chap. 5). Nanos, just as \( \mu \)-patterns have no designated design purpose, but for a different reason. In fact, nanos clutter the design and hide the essence by the details.

It takes a short second to decipher (7) from (8), and to recognize that the snippet (4) is an instance of (1). Programmers are inevitably trained to eliminate clutter and focus on the essence when dealing with language specific, style or convention differences. However, the prevalence of nanos indicates that these small abstract gap accumulate to over a substantial portion of the code.

The purpose of a nano is the small programming task that it represents, and can often be learned from its name. However, this purpose is usually independent of the software in which the nano occurs.

For this reason, the substantial portion of the code dedicated to nanos is noise in many contexts. A contribution of this work is better understanding of the noise. Potential applications of the language are in reducing it.

### 7.2 Automatic Discoverable Idioms

Relevant is the work of Allamanis and Sutton (2014). The authors notion of “idiom”, “a syntactic fragment that recurs frequently across software projects and has a single semantic purpose” is similar to that of nano-patterns. The difference is that nano-patterns are predicates, rather than a single fragment (subject to variations).

Allamanis and Sutton applied machine learning techniques (probabilistic syntactical derivation and nonparametric Bayesian methods) for the first \textit{automatic} discovery of idioms. Idioms thus discovered included “cross-projects idioms that represent important program concepts like object creation, exception handling, etc.”.

Applying the three criteria above, we have that idioms are (by definition) traceable and prevalent. Prevalence of idioms, ranging between 3\% to 31\%, depending on the training and the testing dataset and the fitting parameters, tends be lower than the prevalence of nanos.
To the “purpose” criterion, note that idioms are discovered, while nanos are sought. A priori, discovery does not imply semantic purpose. (Think of compression algorithm which excel in finding recurring sequences, which in most cases, are useless to humans.)

The authors retrospectively demonstrate for some of their idioms semantic purposes such as object creation, exception handling, and resource management. But, in general, purpose and usability are a greater issue with idioms and other recurrences generated by unsupervised machine learning process.

### 7.3 Previous Work on Nano Patterns

The term nano-patterns was first briefly mentioned in the literature by Gil and Maman (2005) Following the work of Høst and Østvold (2008), Singer et al. (2010) gathered a language of 17 nano-patterns, which are demonstrably prevalent (at the 80% level), traceable, and, purposeful. Evidence to purposefulness in Singer et al.’s work is the list of their names: NoParams, NoReturn, Recursive, SameName, Leaf, ObjectCreator, FieldReader, FieldWriter, TypeManipulator, StraightLine, Looping, Exceptions, LocalReader, LocalWriter, ArrayCreator, ArrayReader, ArrayWriter.

Singer et al.’s language and ours are independent: Their language is of properties of methods, where our nanos tend to stand at method, command, expression and field level.\(^1\) Also, the nano-patterns of Singer et al.’s language are essentially orthogonal, while ours are peculiar. Mathematically, this difference means that the combined entropy of their language is greater than ours. As the authors show, this orthogonality makes it possible to define composite nano-patterns capturing correlations between properties.

Similar to the work of Singer et al. is this of Batarseh (2010), in which the author describes a language consists of 16 method properties, Lee et al. (2016), who identified 67 such attributes, organized in three taxa: method signature, method body, and ties of body and behavior. We argue that in difference with languages of orthogonal properties, the “peculiar” nano language presented here are more effective in reducing clutter.

\(^1\)We saved the jargon from the addition of yet another name, “picos”, for patterns at the sub-method level, at the cost of not less fully faithful to the original mention of the term “nano-patterns”. 
Chapter 8

Threats to Validity and Critique

8.1 Limited Domain

Validity of our findings is limited to the domain of the empirical study: OSS Java projects. Generalization to other kinds of software and to other languages is a matter of conjecture. It is likely to find nanos in other programming languages such as C++, C#, Swift, Go. But, different capabilities of languages may drive programmers to prefer some nanos to others, use new nanos specific to a particular language, or abandon some nanos.

The results here do not tell us whether non-OSS code uses the same nanos language as OSS code. More generally, just as there are domain specific pattern languages, there might be domain specific nanos for say, Android application. The present work does resolve this question.

A potential threat to validity is in the use of a biased small corpus Table 3, including the authors’ own code, and other code authors’ domain specific interest in language processing tools. Further, in honesty, the informal refinement process rarely visited projects such as JUnit and hardly ever Atmosphere, manual inspection was disorganized, etc. Consequently, we cannot be statistically confident that our nanos’ yield is exhaustive.

There are two aspects in the bias towards the specific corpus of Table 3: First, manual examination of code snippets before and after nano-transformations was limited to drawn from this corpus. Second, prevalence was computed only in projects from this corpus. The presumption was that prevalent nanos are should maintain their prevalence in spite of the corpus being intentionally biased.
8.2 Missed Nanos and the Language $\mathcal{J}$

More generally, a potential issue of methodologically with this research is that it implicitly made the far reaching (and imprecise) presumption that

“There is one universal prevalent and cohesive language $\mathcal{J}$ of Java nanos”.

(1)

Our informal line of reasoning followed (1), making strong assumptions on the extent of universality and prevalence of $\mathcal{J}$:

1. If $\mathcal{J}$ is prevalent and universal, then we (the authors) being Java programmers ourselves, must have seen many of its nanos. Therefore, even our disorganized brainstorming (Table 4) is likely to hit many (but perhaps not all) of the nanos in $\mathcal{J}$.

2. Similarly, the biased selection of corpus for the initial evolution, and the subjective pruning (Table 4) are rationalized by the presumption that $\mathcal{J}$ universality spans virtually all Java code. Such a universality should be able to sustain this bias.

3. Finally, pruning for prevalence on a somewhat dated baseline corpus (Table 8), makes the presumption of time-universality, i.e., that $\mathcal{J}$ and its prevalence do not change over time.

The contribution made here is much weaker than the presumption (1). All we can claim is:

“There exists a language $\mathcal{J}'$ of Java nanos which is universal (under some definition of universality), prevalent (under some definition of prevalence) over OSS projects, and, cohesive (i.e., tree organized)”.

(2)

Presumptions such as (1) are too vague to a matter of proof. But, the result (2) is an indication to the truth in presumption. The threat to validity is that the attempt to produce the result (2), the evolved language $\mathcal{J}'$, under estimates $\mathcal{J}$ (i.e., inferior recall). The answer to this threat, is in the high prevalence of $\mathcal{J}'$ over two independently selected large corpora is substantial (second and third sections of Table 2), and not too inferior than prevalence of $\mathcal{J}'$ over the baseline corpora on which $\mathcal{J}'$ was evolved.

Further research is be to improve the prevalence values by specifically looking for new candidates in other corpora. Also, the nano-tracer capabilities are limited: it uses minimal data flow analysis heuristics, it does not create a
control flow graph, its analysis of the namespace is crude, etc. This weakness is a strength of each of the nanos in $J'$: we know now that a light weight tracer is sufficient to show their prevalence. This weakness means also that $J'$ may be incomplete: prevalence of nano candidates such as iteration over pairs ($D_4$ in Table 4) could be demonstrated with a more powerful tracer.
Chapter 9

Nanos Detection Analysis

9.1 The Spartanizer

Refactoring is the practice of altering the structure of the code without changing its observable behavior, in the sake of improving design, and in particular, extensibility and modularity (Fowler, 1999)(Opdyke and Johnson, 1990).

The Spartanizer\(^1\), an interactive Eclipse plugin, which currently offers over 150 refactorings techniques whose application is geared towards achieving the end of Spartan programming. Spartan programming (Gil, 2010; Gil and Orrú, 2017) is a methodology and coding style which, in the spirit of visionaries like Dijkstra (Dijkstra, 1968), focuses on minimalism. Spartan code makes frugal use of code elements: lines, characters, arguments, nesting, use of conditional and iterative control, etc. Spartan code is the software equivalent of laconic speech: saying the most in fewest words.

The Spartanizer works with a toolbox equipped with tippers. A tipper is a reification of transformation rule, specifically, a (nano) refactoring technique, or more specifically, a generator of spartanization tips. This modular structure makes it possible to easily add tippers: which are just classes that implement the Tipper protocol. The first version of the plugin (circa 2013) had six tippers. Their number increased to over 150 as of print.

A special characteristic of the Spartanization process is that of bringing code to a normalized form, it eliminates verbosity revealing the pure intentions of the program. Hence, Spartanization contributes to the process of finding nanos. Our analysis spartanizes the code and then mines nanos out of it using special tippers. Whenever a code snippet is detected as a nano, it is refactored to an idiomatic expression/statement marking the existence of the

\(^1\)https://github.com/SpartanRefactoring/Spartanizer
nano, so it would not be detected twice.

9.2 Leonidas

9.2.1 Motivation

Implementing tippers for nanos detection requires writing cumbrous code, traversing an AST (Abstract Syntax Tree, constructed of ASTNodes), inquiring it about its structure (is the ASTNode a ForStatement? an InfixExpression?, etc.) and contents. To ease the implementation, several libraries were implemented, supplying type inquiry, type casting, null-proof navigation, nodes construction, nodes filtering, and more. Yet, implementing even a simple nano as defaultsTo might be tedious and require long code. The problem escalates when the nano appears in several forms, what requires programming the tipper to deal with all the different cases.

9.2.2 Solution

Leonidas is a high abstraction library implemented by the authors, allowing programmers to construct tippers out of a pattern string and a replacement string. The tipper generates an AST out of the pattern string, and given an ASTNode to examine, it compares its structure to the AST. If matches, the tipper can replace the ASTNode with an AST generated out of the replacement string.

Leonidas defines a language of wildcard symbols that may be contained in strings and match an ASTNode of some type or a set of values, for example $X$ matches an Expression. The full language is depicted in Table 7

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>List of a MethodInvocation arguments</td>
</tr>
<tr>
<td>$B$</td>
<td>Block</td>
</tr>
<tr>
<td>$D$</td>
<td>Default value (null, false, 0, 0., 0L)</td>
</tr>
<tr>
<td>$L$</td>
<td>Literal</td>
</tr>
<tr>
<td>$M$</td>
<td>MethodInvocation</td>
</tr>
<tr>
<td>$N$</td>
<td>Name</td>
</tr>
<tr>
<td>$T$</td>
<td>Type</td>
</tr>
<tr>
<td>$X$</td>
<td>Expression</td>
</tr>
<tr>
<td>$SN$</td>
<td>SimpleName</td>
</tr>
</tbody>
</table>

Table 7: The Leonidas language
Constructing `tippers` has never been so easy. For example, one case of the `defaultsTo` nano is implemented using the simple snippet:

```java
pattern("$X1 !== null ? $X1 : $X2").replaceWith("defaults($X1).to($X2)");
```

A case of the sub-nano `allHolds` is implemented using:

```java
pattern("for($T,$N1:$X1) if($X2) return false; return true;")
.replaceWith("return $X1.stream().allMatch($N1 -> !($X2))");
```

(1) (2)
Chapter 10
Conclusion

We introduced a universal taxonomized language of 38 nano-patterns for Java. A nano pattern is a recurring solution to a common task, such as safeguarding against null, adding default arguments, or simple stub implementation. Nanos are often clutter of small implementation level code, and are likened to nuts and bolts in the software world. Applications are:

- Language design, library design, macros and syntax sugaring may reduce the clutter of low level details, by eliminating the nuts and bolts or making them less conspicuous.

- Conversely, the contribution has its use in visualization, reverse engineering, code summarization, and, refactoring. In these, the challenge is to hide, not eliminate, the substantial accumulative nuts-and-bolts clutter that nanos represent.

- More generally, applications may follow the applicative research finding unexpected code aspects which link with recurrences, e.g., if the presence of nano-patterns is correlated with bug proneness, then, the easy to detect presence of nanos might be used as predictor of the illusive bug proneness property.

A (probable) unicum in the literature is the description of our nano harvesting including candidate collection and pruning. We dedicated some attention to explaining the rationale behind this process, and pointed out applications in other domains of the ideas behind the r-index and the h-index on which it is based.

The subjective and sample bias in the harvest may have lead us to miss some patterns. However, this bias, together with the prevalence tests, strengthens the evidence of universality of the language we found, at least
for its particular domain of OSS written in Java. The taxonomy we describe evolved over the nanos we found. A crucial test of the validity of the taxonomy would be in applying it to these nanos that we missed.

We provide evidence on the substantial prevalence of nano-pattern in actual code. About half of the methods are an instance of nanos; the third of all remaining methods have one nano or more. A third of the commands and quarter of expressions are nano instances. Also, prevalence of our subjectively selected language, rivals that of patterns discovered by an automatic learning process.

A contribution is also our OSS nano-tracer.
Appendix A

Other Related Work

**Vocabulary vs. Structure** In Chap. 2 we divided this research by theme: *generative, applicative* and *opportunistic*. Here we divide research into two the study of the nomenclature—the *vocabulary* that programmers tend to choose, and the study of the recurrence in syntactical *structure* of programs, as we do here.

Studies of nomenclature include the work of Linstead et al. (2009), who analyzed the source code vocabulary, identifying trends specific to syntactic kinds such as classes and interfaces. Enslen et al. (2009)’s algorithm for splitting identifiers into words’ sequences makes use of word frequencies in source code to optimize the split. Abebe et al. (2009) investigate the evolution of vocabulary with time. See also the work of Newman et al. (2017) and the bibliography there.

Høst and Østvold (2008) deal with the problem of generating a *semantics which captures our common interpretation of method names*. They started from the definition of *traceable pattern* and proposed a set of *traceable attributes* useful as a building blocks to define nano-patterns. They explicitly claim that they are not proposing nano-patterns. In a previous work the same authors analyzed the implementation of methods taken from a corpus of Java applications, in order to determine which word to use as a method name (Høst and Østvold, 2007).

Andras et al. (2013); Moreno and Marcus (Moreno and Marcus) investigated the consistency between design and implementation of OO methods using a run-time analysis of their behavior. They compared the outcome of the run-time with their stereotype (Andras et al., 2013). Stereotypes capture the intent of a method/class and are considered as an extension of the micro/nano patterns concept (Dragan et al., 2006). This idea is often used in automatic summarization applications.

Høst and Østvold (2009) also take interest in vocabulary, but they match
it against structure. In noticing that method names reflect patterns in the code’s structure, the authors devise automatic tools to identify “naming bugs” and for suggesting more appropriate names for methods. Also, Kashiwabara et al. (2014) and Kashiwabara et al. (2015) authors present a technique to identify candidate verbs to be used as method names. There is room for similar research that links structure and vocabulary in the context of nanos, i.e., linking these with names, or, conversely, eliminating these in making method names.

Attempts to employ recurring structure for software engineering purposes can be traced back to the work on beacons (Brooks, 1983; Crosby et al., 2002), which are stereotypical (and hence recurring) and telling segments of code that are quick to capture by experienced programmers.

Qiu et al. (2017) gave an empirical support to structure based search for recurring patterns. In particular, the authors showed that the use of syntactical rules in actual programs follows a Zipf like distribution (just as the use words in natural language), or, conversely, that a small set of syntactic structures tend to govern entire projects.

**Hypothesis Based vs. Automatic Search** The identification of repeating patterns, idioms, common vocabulary etc., like is traditionally done in one of two ways: scholars’ hypothesis confirmed by automatic discovery and/or manual browsing and search (see Gamma et al., 1995; Singer et al., 2010; Gil and Maman, 2005; Batarseh, 2010), and, a hypothesis-free, automatic search (e.g., see Linstead et al., 2009; Spinellis and Raptis, 2000; Sutton et al., 2010; Guéhéneuc et al., 2010; Allamanis and Sutton, 2014).

The evolution of our language is hypothesis guided, but, unlike previous work, we describe also cases when the hypothesis was rejected. Chap. 5 reported on 82 nano “candidates”, organized in 14 preliminary taxa. Some of these candidates matured to be nanos, others were rejected for not being sufficiently prevalent, or for being difficult to trace with our tools.

**Idioms vs. Patterns** The more popular literature speaks of idioms. There are online catalogs of commendable idioms, e.g., (Wikibooks, 2017; Editor, 2017; Chuan, 2014; Waldron, 2014), IDEs provide support to idioms definition and customization (Recommenders-Contributors, 2014; JetBrains, 2014), and books (Gurewich and Gurewich, 1996; Laffra, 1996; Larman and Guthrie, 1999) which try to teach style and idioms to beginners. In these, idioms are rarely defined, but they are often meant to be tips of programming, mostly at the class and below levels.

In the scientific literature on idioms we mention the work of Sutton et al. (2010) who studied idioms in generic C++ libraries finding a high degree (circa 85%) of “idiomization”, i.e., coverage by \( \mu \)-patterns (Gil and Maman, 2005) (which can be thought idioms at the class level). Other studies of
idioms can be found, e.g., here (Allamanis and Sutton, 2014; Koenig, 1995; Langer, 2001; Wills, 1990).

**Patterns on Iterative Structures** Abd-El-Hafiz and Basili (1996) present a classification of loops by complexity levels. Wang et al. (2015) present an automatic approach to identify the high level action implemented by a given loop. Also, related is the work of Allamanis et al. (2016). In general, our loop analysis tools are weaker than these.

**Taxonomy of nanos** Mens et al. (2002), proposed to define patterns with a declarative meta programming language (DML), which is a Prolog variant. This approach can be used to build development tools to support developers in their programming tasks. The authors did not provide a catalog, but illustrated the use of DML by applying it to some of the best practices of Smalltalk programmers, design patterns, such as *visitor*, heuristics and code smells like *duplicated code*. 

53
Appendix B

Nano Snippets

B.1 Elaborators


```java
if (!doubleIsDifferent(unexpected, actual, delta)) failEquals(message, Double.valueOf(actual));
```

(project: Junit, file: Assert.java)

whenHoldsOn.Composite.Command

```java
if (binding instanceof ConstructorBindingImpl) {
    return ((ConstructorBindingImpl) binding).getInternalDependencies();
} else if (binding instanceof HasDependencies) {
    return ((HasDependencies) binding).getDependencies();
} else {
    return ImmutableSet.of();
}
```

(Guice, InjectorImpl.java)

defaultsTo.Weak.Elaborator.Expression

```java
e.getCause() == null ? e : e.getCause()
```

(Cucumber-Jvm, RhinoStepDefinition.java)

questionQuestion.Weak.Elaborator.Expression

```java
keyValueWriterClass == null ? new HadoopStoreWriter() : (KeyValueWriter)Utils.callConstructor(keyValueWriterClass)
```

(Voldemort, HadoopStoreBuilderReducer.java)

    newSearcher != null && closeNewSearcher

(ElasticSearch, InternalEngine.java)

evaluateUnlessDefaultsTo.Weak.Elaborator.Expression

    this.markedWriterIndex <= decrement ? 0 : this.markedWriterIndex - decrement

(Netty, AbstractByteBuf.java)

safeCast.Weak.Elaborator.Expression

    (ThreadPoolExecutor)holder.executor

(ElasticSearch, ThreadPool.java)

throwOnFalse.Weak.Elaborator.Expression

    if (!buffer.isReadable()) throw new EOFException();

(Netty, ByteBufInputStream.java)

throwOnNull.Weak.Elaborator.Expression

    if (shellBuilder == null) throw new AssertionError("Already built, builders are not reusable.");

(Guice, InternalInjectorCreator.java)

NotNullRequired.Atomic.Command

    private static Class<?>[] directCategories(Description description) {
        if (description == null) {
            return new Class<?>[0];
        }
        ...
    }

(Junit, Categories.java.java)

NotNullAssumed.Atomic.Command

    if (currentMapper == null) return super.newTermQuery(x);

(ElasticSearch, MapperQueryParser.java)
holdsOrReturn.Atomic.Command

if (environment == null) return;

(Titan, BerkeleyJEStoreManager.java)


try {
    Class<?> clazz = GUICE_CLASS_LOADER.loadClass(name);
    if (resolve) {
        resolveClass(clazz);
    }
    return clazz;
} catch (Throwable e) {
}

(Guice, BytecodeGen.java)

letInNext.Elaborator.Composite.Command

String methodName=each.getMethodName();
return (methodName == null ? Request.aClass($) : Request.method($,methodName).getRunner();

(Junit, MaxCore.java)

B.2 Multitude

forFromTo.Iterative.Composite.Command

for (int x=0; x < counter; ++x) $.append("\t");

(ElasticSearch, ExceptionsHelper.java)

aggregate.Multitude.Expression

for (JRubyWorldDefinition w : worldDefinitions) currentWorld=w.execute(
    currentWorld);

(Cucumber-Jvm, JRubyBackend.java)

selectBy.Select.Multitude.Expression

for (Object a : args) jrubyArgs.add(a == null ? null : !(a instanceof DataTable) ? stepdef.getRuntime().newString((String)a) : JavaEmbedUtils.javaToRuby(stepdef.getRuntime(),a));

(Cucumber-Jvm, JRubyBackend.java)
forEach.Iterative.Composite.Command

```java
for (String columnName : columnNames) let(() -> fieldNames.indexOf((new CamelCaseStringConverter()).map(columnName)).in(index -> ![n++=index == 
-1 ? "" : fieldValues.get(index));
```

(Cucumber-Jvm, ComplexTypeWriter.java)

firstSuchThat.Select.Multitude.Expression

```java
for (int x=0; x < output2.length; ++x) if (output2.bytes[x + output2.offset]
== payloadSep) {
    sepIndex=x;
    break;
}
```

(ElasticSearch, XAnalyzingSuggester.java)

B.3 Phrase

myArguments.Phrase

```java
@Override public boolean hasBankAccount(String bank,String name){
    return this.BOSEconomy.isBankOwner(bank,name) || this.BOSEconomy.
    isBankMember(bank,name);
}
```

(Essentials, BOSE7.java)

myName.Phrase

```java
protected String uuid(AtmosphereResource r) {
    return r.uuid();
}
```

(atmosphere, UUIDBroadcasterCache.java)

B.4 Field

cost.Atomic.Field

```java
public static final String FACTOR_SYS_PROP="ts.timeout.factor";
```

(WildFly, TimeoutUtil.java)
setter.Elaborator.Field

```java
public void setName(String name){
    this.name=name;
}
```

(WildFly, ManagedCreateLdapServer.java)

getter.Elaborator.Field

```java
public String getProductName(){
    return name;
}
```

(WildFly, ProductConfig.java)

fluentSetter.Elaborator.Field

```java
public Builder setNumericType(NumericType x){
    this.numericType=x;
    return this;
}
```

(ElasticSearch, PackedArrayIndexFieldData.java)

vanillaCollection.Atomic.Field

```java
private final List<List<String>> arguments=new ArrayList<List<String>>() {
```

(WildFly, Usage.java)

B.5 Sink Method

constantValue.Atomic.Method

```java
@override protected String getOutputSuffix(){
    return "</div>";
}
```

(Docx4j, SvgConversionContext.java)

thrower.Atomic.Method

```java
public long time(){
    throw new UnsupportedOperationException();
}
```

(ElasticSearch, VersionValue.java)
defaultValue.Atomic.Method

```java
@Override public Permission getRequiredPermission(){
    return null;
}
```

(Hazelcast, AuthenticationRequest.java)

examiner.Atomic.Method

```java
public boolean shouldReply(){
    return !noreply;
}
```

(Hazelcast, TouchCommand.java)

letIn.Atomic.Method

```java
public static long addJitter(final long pause, final float jitter){
    long $=pause + jitter * (long)pause * (RANDOM.nextFloat() - 0.5f);
    return $ > 0 ? $ : 1;
}
```

(Hbase, ConnectionUtils.java)

B.6 Delegating Method

factory.Elaborator.Method

```java
protected AsyncSupportResolver createAsyncSupportResolver(){
    return new DefaultAsyncSupportResolver(config);
}
```

(Atmosphere, AtmosphereFramework.java)

defaultArguments.Elaborator.Method

```java
public static void validate(String s,Type t){
    validate(s,t,false);
}
```

(Atmosphere, UriComponent.java)

adjuster.Elaborator.Method

```java
public void trackSynchronously(FocusPoint x){
    httpRequest.request(urlBuildingStrategy.buildURL(x));
}
```

(Atmosphere, JGoogleAnalyticsTracker.java)
delegator.Elaborator.Method

```java
@Override public Action doService(AtmosphereRequest r, AtmosphereResponse res)
        throws IOException, ServletException {
    return super.service(r, res);
}
```

(Atmosphere, Tomcat7BIOSupportWithWebSocket.java)

B.7 Constructor

pojo.Atomic.Constructor

```java
private WriteResult(AtmosphereResource r, Object message) {
    this.r = r;
    this.message = message;
}
```

(Atmosphere, JSR356WebSocket.java)

super.Elaborator.Constructor

```java
public WebLogicServlet30WithWebSocket(AtmosphereConfig config) {
    super(config);
}
```

(Atmosphere, WebLogicServlet30WithWebSocket.java)

this.Elaborator.Constructor

```java
public AtmosphereServlet(boolean isFilter) {
    this(isFilter, true);
}
```

(Atmosphere, AtmosphereServlet.java)
Appendix C

Reproducibility Information

Table 8: Baseline corpus of 26 OSS Java projects: identification, reproducibility information and work volume indicators.

<table>
<thead>
<tr>
<th>Project</th>
<th>First Version</th>
<th>Last Version</th>
<th>Last Hash</th>
<th>#Days</th>
<th>#Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmosphere</td>
<td>10-04-30</td>
<td>14-04-28</td>
<td>557e1044</td>
<td>1,459</td>
<td>62</td>
</tr>
<tr>
<td>CraftBukkit</td>
<td>11-01-01</td>
<td>14-04-23</td>
<td>62ca8158</td>
<td>1,208</td>
<td>156</td>
</tr>
<tr>
<td>Cucumber-jvm</td>
<td>11-06-27</td>
<td>14-07-22</td>
<td>fd764318</td>
<td>1,120</td>
<td>93</td>
</tr>
<tr>
<td>Docx4j</td>
<td>12-05-12</td>
<td>14-07-04</td>
<td>8edaddfa</td>
<td>783</td>
<td>19</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>11-10-31</td>
<td>14-06-20</td>
<td>812972ab</td>
<td>963</td>
<td>129</td>
</tr>
<tr>
<td>Essentials</td>
<td>11-03-19</td>
<td>14-04-27</td>
<td>229f9f0</td>
<td>1,134</td>
<td>67</td>
</tr>
<tr>
<td>Guava</td>
<td>11-04-15</td>
<td>14-02-25</td>
<td>6fda5f06</td>
<td>1,047</td>
<td>12</td>
</tr>
<tr>
<td>Guice</td>
<td>07-12-19</td>
<td>14-07-01</td>
<td>76be88e8</td>
<td>2,386</td>
<td>17</td>
</tr>
<tr>
<td>Hadoop-common</td>
<td>11-08-25</td>
<td>14-08-21</td>
<td>0c648ba0</td>
<td>1,092</td>
<td>69</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>09-07-21</td>
<td>14-07-05</td>
<td>3c4bc794</td>
<td>1,809</td>
<td>65</td>
</tr>
<tr>
<td>Hbase</td>
<td>12-05-26</td>
<td>14-01-30</td>
<td>e67682c8</td>
<td>613</td>
<td>25</td>
</tr>
<tr>
<td>Hector</td>
<td>10-12-05</td>
<td>14-05-28</td>
<td>f2fe542c</td>
<td>1,270</td>
<td>95</td>
</tr>
<tr>
<td>Hibernate-orm</td>
<td>09-07-07</td>
<td>14-07-02</td>
<td>b0a2ae9d</td>
<td>1,821</td>
<td>150</td>
</tr>
<tr>
<td>Jclounds</td>
<td>09-04-28</td>
<td>14-04-25</td>
<td>f1a0370b</td>
<td>1,823</td>
<td>100</td>
</tr>
<tr>
<td>Jna</td>
<td>11-06-22</td>
<td>14-07-07</td>
<td>a5942aaaf</td>
<td>1,110</td>
<td>46</td>
</tr>
<tr>
<td>Junit</td>
<td>07-12-07</td>
<td>14-05-03</td>
<td>56a03468</td>
<td>2,338</td>
<td>91</td>
</tr>
<tr>
<td>K-9</td>
<td>08-10-28</td>
<td>14-05-04</td>
<td>95f33c38</td>
<td>2,014</td>
<td>81</td>
</tr>
<tr>
<td>Lombok</td>
<td>09-10-14</td>
<td>14-07-01</td>
<td>3c4f6841</td>
<td>1,721</td>
<td>22</td>
</tr>
<tr>
<td>Mongo-java-driver</td>
<td>09-01-08</td>
<td>14-06-16</td>
<td>5565c46e</td>
<td>1,984</td>
<td>75</td>
</tr>
<tr>
<td>Netty</td>
<td>11-12-28</td>
<td>14-01-28</td>
<td>3061b154</td>
<td>762</td>
<td>72</td>
</tr>
<tr>
<td>Openmrs-core</td>
<td>10-08-16</td>
<td>14-06-18</td>
<td>05292d98</td>
<td>1,401</td>
<td>119</td>
</tr>
<tr>
<td>RxJava</td>
<td>13-01-23</td>
<td>14-04-25</td>
<td>12723ef0</td>
<td>456</td>
<td>47</td>
</tr>
<tr>
<td>Spring-framework</td>
<td>10-10-25</td>
<td>14-01-28</td>
<td>e5f908b1</td>
<td>1,190</td>
<td>53</td>
</tr>
<tr>
<td>Titan</td>
<td>13-01-04</td>
<td>14-04-17</td>
<td>55de01a3</td>
<td>468</td>
<td>17</td>
</tr>
<tr>
<td>Voldemort</td>
<td>01-01-01</td>
<td>14-04-28</td>
<td>fb3203f3</td>
<td>4,865</td>
<td>56</td>
</tr>
<tr>
<td>Wildfly</td>
<td>10-06-08</td>
<td>14-04-22</td>
<td>5a29e860</td>
<td>1,413</td>
<td>194</td>
</tr>
</tbody>
</table>

Total: 26 projects 38,250 1,932

*Highlighted projects were included in the biased corpus (Table 3).
Table 9: Reproducibility information for the Java corpus consisting of the 26 most starred Android projects on GitHub

<table>
<thead>
<tr>
<th>Project</th>
<th>Commit Id</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>bazel</td>
<td>5dbb23b</td>
<td>Fri Feb 24 17:14:35 2017</td>
</tr>
<tr>
<td>disruptor</td>
<td>2314a27</td>
<td>Mon Nov 21 20:16:47 2016</td>
</tr>
<tr>
<td>dropwizard</td>
<td>e823137</td>
<td>Fri Feb 24 19:55:30 2017</td>
</tr>
<tr>
<td>druid</td>
<td>78b8a57</td>
<td>Sat Feb 25 11:30:30 2017</td>
</tr>
<tr>
<td>fastjson</td>
<td>490213c</td>
<td>Tue Feb 21 05:04:12 2017</td>
</tr>
<tr>
<td>gradle</td>
<td>2f9dac0</td>
<td>Sun Feb 26 08:06:12 2017</td>
</tr>
<tr>
<td>Hystrix</td>
<td>c674545</td>
<td>Fri Feb 10 21:05:15 2017</td>
</tr>
<tr>
<td>java-design-patterns</td>
<td>cca4760</td>
<td>Sun Feb 12 00:33:30 2017</td>
</tr>
<tr>
<td>jedis</td>
<td>2c6ade7</td>
<td>Wed Feb 22 09:43:01 2017</td>
</tr>
<tr>
<td>kafka</td>
<td>5b682ba</td>
<td>Sun Feb 26 04:44:32 2017</td>
</tr>
<tr>
<td>kotlin</td>
<td>fa5728b</td>
<td>Fri Feb 24 17:58:25 2017</td>
</tr>
<tr>
<td>leakcanary</td>
<td>f0bfbbd</td>
<td>Sun Feb 26 10:12:58 2017</td>
</tr>
<tr>
<td>metrics</td>
<td>6b6ae6f</td>
<td>Fri Feb 24 20:05:34 2017</td>
</tr>
<tr>
<td>mockito</td>
<td>fad7211</td>
<td>Sun Feb 26 01:25:29 2017</td>
</tr>
<tr>
<td>mybatis-3</td>
<td>cd2cc17</td>
<td>Sat Feb 25 04:04:18 2017</td>
</tr>
<tr>
<td>neo4j</td>
<td>8ae7eac</td>
<td>Fri Feb 24 23:34:00 2017</td>
</tr>
<tr>
<td>okhttp</td>
<td>89621df</td>
<td>Wed Feb 15 05:53:53 2017</td>
</tr>
<tr>
<td>presto</td>
<td>0e6c3b2</td>
<td>Sat Feb 25 03:40:27 2017</td>
</tr>
<tr>
<td>retrofit</td>
<td>39fcd6b4</td>
<td>Fri Feb 24 19:51:33 2017</td>
</tr>
<tr>
<td>spark</td>
<td>9618b83</td>
<td>Thu Feb 23 10:58:28 2017</td>
</tr>
<tr>
<td>spring-boot</td>
<td>3a8be10</td>
<td>Fri Feb 24 18:46:04 2017</td>
</tr>
<tr>
<td>storm</td>
<td>cd116e</td>
<td>Fri Dec 13 20:53:13 2013</td>
</tr>
<tr>
<td>webmagic</td>
<td>a87ea64</td>
<td>Sat Feb 25 16:46:29 2017</td>
</tr>
<tr>
<td>zipkin</td>
<td>4624229</td>
<td>Wed Feb 15 16:45:00 2017</td>
</tr>
</tbody>
</table>
Table 10: Reproducibility information for Android corpus consisting of the 26 most starred Android projects on GitHub

<table>
<thead>
<tr>
<th>Project</th>
<th>Commit Id</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>androidannotations</td>
<td>1e91a3b</td>
<td>Sun Feb 26 16:56:35 2017</td>
</tr>
<tr>
<td>Android-CleanArchitecture</td>
<td>ae30300</td>
<td>Mon Jan 9 00:30:18 2017</td>
</tr>
<tr>
<td>Android-Universal-Image-Loader</td>
<td>4c72879</td>
<td>Tue Jan 26 13:29:19 2016</td>
</tr>
<tr>
<td>AndroidUtilCode</td>
<td>a2e6a93</td>
<td>Mon Feb 27 13:33:55 2017</td>
</tr>
<tr>
<td>butterknife</td>
<td>581666a</td>
<td>Tue Jan 24 18:57:50 2017</td>
</tr>
<tr>
<td>fresco</td>
<td>9bf8cf5</td>
<td>Fri Feb 24 22:04:35 2017</td>
</tr>
<tr>
<td>glide</td>
<td>3bcea91</td>
<td>Wed Feb 15 18:32:15 2017</td>
</tr>
<tr>
<td>HomeMirror</td>
<td>a71c860</td>
<td>Fri May 20 21:54:56 2016</td>
</tr>
<tr>
<td>libgdx</td>
<td>6302af8</td>
<td>Mon Feb 27 02:18:33 2017</td>
</tr>
<tr>
<td>lottie-android</td>
<td>09f459c</td>
<td>Sat Feb 25 04:17:52 2017</td>
</tr>
<tr>
<td>Material-Animations</td>
<td>28e65b2</td>
<td>Sun May 15 20:30:33 2016</td>
</tr>
<tr>
<td>material-dialogs</td>
<td>2205c69</td>
<td>Tue Feb 21 17:50:45 2017</td>
</tr>
<tr>
<td>MPAndroidChart</td>
<td>0c919ab</td>
<td>Wed Feb 22 17:29:33 2017</td>
</tr>
<tr>
<td>PhotoView</td>
<td>6c227ee</td>
<td>Sun Dec 18 02:13:52 2016</td>
</tr>
<tr>
<td>picasso</td>
<td>0b38d119</td>
<td>Fri Feb 24 17:18:20 2017</td>
</tr>
<tr>
<td>plaid</td>
<td>abf8669</td>
<td>Thu Feb 23 15:34:32 2017</td>
</tr>
<tr>
<td>PocketHub</td>
<td>a43a6f</td>
<td>Mon Feb 27 10:54:37 2017</td>
</tr>
<tr>
<td>RxAndroid</td>
<td>eb27d75</td>
<td>Mon Jan 9 20:59:49 2017</td>
</tr>
<tr>
<td>SlidingMenu</td>
<td>4254fec</td>
<td>Sun Mar 9 23:30:35 2014</td>
</tr>
<tr>
<td>tinker</td>
<td>9d2b250</td>
<td>Tue Jan 17 05:47:08 2017</td>
</tr>
<tr>
<td>ViewPagerIndicator</td>
<td>8cd549f</td>
<td>Wed Sep 12 19:08:57 2012</td>
</tr>
<tr>
<td>weex</td>
<td>4733936</td>
<td>Mon Feb 27 07:32:23 2017</td>
</tr>
</tbody>
</table>
Table 11: Values of the $r_-$, $r_i-$ and $r_x-$ indices in the baseline-corpus (Table 8) at the end of the monitored period, and differences from initial values

<table>
<thead>
<tr>
<th>Project</th>
<th>$r$</th>
<th>$r_i$</th>
<th>$r_x$</th>
<th>$r_i - r_x$</th>
<th>$\Delta r$</th>
<th>$\Delta r_i$</th>
<th>$\Delta r_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmosphere</td>
<td>34</td>
<td>25</td>
<td>22</td>
<td>3</td>
<td>17</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>CraftBukkit</td>
<td>42</td>
<td>39</td>
<td>17</td>
<td>22</td>
<td>41</td>
<td>38</td>
<td>16</td>
</tr>
<tr>
<td>Cucumber</td>
<td>16</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Docx4j</td>
<td>54</td>
<td>46</td>
<td>28</td>
<td>18</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>63</td>
<td>59</td>
<td>27</td>
<td>32</td>
<td>34</td>
<td>31</td>
<td>12</td>
</tr>
<tr>
<td>Essentials</td>
<td>41</td>
<td>29</td>
<td>27</td>
<td>2</td>
<td>11</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Guava</td>
<td>33</td>
<td>32</td>
<td>13</td>
<td>19</td>
<td>17</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Guice</td>
<td>25</td>
<td>20</td>
<td>16</td>
<td>4</td>
<td>15</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Hadoop-common</td>
<td>99</td>
<td>90</td>
<td>47</td>
<td>43</td>
<td>67</td>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>52</td>
<td>48</td>
<td>20</td>
<td>28</td>
<td>17</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Hbase</td>
<td>89</td>
<td>72</td>
<td>47</td>
<td>25</td>
<td>56</td>
<td>44</td>
<td>26</td>
</tr>
<tr>
<td>Hector</td>
<td>20</td>
<td>17</td>
<td>12</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Hibernate-orm</td>
<td>83</td>
<td>75</td>
<td>35</td>
<td>40</td>
<td>18</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>Jclouds</td>
<td>47</td>
<td>39</td>
<td>29</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Jna</td>
<td>26</td>
<td>22</td>
<td>14</td>
<td>8</td>
<td>14</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>Junit</td>
<td>15</td>
<td>14</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>K-9</td>
<td>46</td>
<td>42</td>
<td>28</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Lombok</td>
<td>36</td>
<td>30</td>
<td>21</td>
<td>9</td>
<td>20</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>Mongo-java-driver</td>
<td>25</td>
<td>23</td>
<td>10</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Netty</td>
<td>37</td>
<td>35</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Openmrs-core</td>
<td>65</td>
<td>52</td>
<td>43</td>
<td>9</td>
<td>42</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>RxJava</td>
<td>17</td>
<td>15</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>-6</td>
</tr>
<tr>
<td>Spring-framework</td>
<td>63</td>
<td>59</td>
<td>29</td>
<td>30</td>
<td>13</td>
<td>16</td>
<td>-3</td>
</tr>
<tr>
<td>Titan</td>
<td>28</td>
<td>23</td>
<td>19</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Voldemort</td>
<td>52</td>
<td>45</td>
<td>31</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Wildfly</td>
<td>106</td>
<td>94</td>
<td>46</td>
<td>48</td>
<td>63</td>
<td>59</td>
<td>24</td>
</tr>
</tbody>
</table>

For each project the table reports $r$, $r_i$, $r_x$, the difference between the two and the differences between $r$, $r_i$, $r_x$ values in an initial version of the project and the latest one.
Table 12: The 5 sub-nanos augmenting the language of Table 1

<table>
<thead>
<tr>
<th>(*)</th>
<th>a(*)</th>
<th>b(*)</th>
<th>c</th>
<th>Name</th>
<th>Intent $^d$</th>
</tr>
</thead>
</table>
| 1   | $G_2$ | 12%  | ✓ | anyHolds | \(\exists x \in M, p(x)\)  
    |      |      |    |        | \(M\text{-anyMatch}(\cdot)\) |
| 2   | $G_4$ | 0%   | ✓ | allHolds | \(\forall x \in M, p(x)\)  
    |      |      |    |        | \(M\text{-allMatch}(\cdot)\) |
| 3   | $H_2$ | 0%   | ✓ | countIf | \(\#\{x \in M, p(x)\}\)  
    |      |      |    |        | \(M\text{-filter}(\cdot).\text{count}()\) |
| 4   | $H_5$ | 23%  | ✓ | forEachSuchThatDo | \(\forall x \in M, p(x): C\)  
    |      |      |    |        | \(M\text{-filter}(\cdot).\text{foreach}(\cdot)\) |
| 5   | $H_7$ | 19%  | ✓ | aggregateWith | \(\text{Aggregate } M\text{ [st } p(\cdot)]\)  
    |      |      |    |        | \(\text{with } A(\cdot, \cdot)\)  
    |      |      |    |        | \(M\text{-filter}(\cdot).\text{reduce}(\cdot)\) |

$^a$ Candidate(s) from which this nano was created  
$^b$ Prevalence score in the baseline corpus  
$^c$ Can be emulated by lazy-evaluation, using lambda functions  
$^d$ e.g., pseudo-code, Java 8 streams, suggestion for extending plain Java syntax, etc.
Appendix D

Java Like Stenography

We allow method declaration to include stenography extension for some of the nano-patterns:

1. A method may have a body even if ends with “;”, provided that the method declaration defines how the function return value (or body in case of void function) is implemented.

2. After the method header one may add = e, which means that the function’s return value is computed by evaluating this expression. This specification replaces the function’s body.

3. After the method header one may add → g, which implies that the function’s return value is computed by delegating to function g which is declared elsewhere.

4. The formal arguments list may be interspersed with expressions, which depend on other real formal arguments.

5. A formal argument definition may be written as:
   ```java
   Double pow(double d → Double.valueOf(d), int n → Integer.valueOf(n));
   ```
Bibliography


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


75


Laura Moreno and Andrian Marcus. JStereoCode: automatically identifying method & class stereotypes in Java code.


77


Peter Van Emde Boas. 1997. 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. citeseer.ist.psu.edu/vanemdeboas97resistance.html


לא ניתן לקרוא את התוכן שלפה בפורמט המוצג בתמונה.
תקציר

בעבודה זו נמדדים מתיחות מתח גבוה באמצעות גרף גרפי משטרת של גרף גרפי משטרת של גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף גרף Гр
המחקرق בנצי ברכהיון של פרופסור יוסי גל, בכ Reco למדעי המחשב.

חלק מהхранואת בតבון: הפרסומיים המאמרים שכתב והות.itemView למחקרים בכ Reco למדעי הבכירות


בנוסף, התחפשת הונהג כמאמר לכת.

תודה

אני מודה ל antic רע החיפה הניסיון הנדיב הבכירות בבב_extraction.
נירב עueblo

לשם מדלי חלקי שלeldaדות לחבלת החותר
נגיסטר למ נוספים במגורי המתחב

אורי מרקוביץ'

ורגב לסט הכותבי - סוכן טכנולוגי לישראלי
尼斯ח הכתשעל בהפת אפריל 2017
שפת נורתברניית ל׳יאוה

אורי מרקוביץ'