A Primer on High-Quality Identifier Naming

Anthony Peruma
Assistant Professor
Information and Computer Sciences Department
University of Hawaiʻi at Mānoa

Christian Newman
Assistant Professor
Department of Software Engineering
Rochester Institute of Technology

UNIVERSITY of HAWAIʻI MĀNOA

RIT | Rochester Institute of Technology

About Anthony...

Experience/Qualifications

Assistant Professor
University of Hawaiʻi at Mānoa, USA

Ph.D. in Computing and Information Sciences
Rochester Institute of Technology, USA

Masters in Software Engineering
Rochester Institute of Technology, USA

10+ years of industry experience

Research Interests

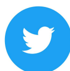
Program Comprehension - Identifier Naming

Software Quality - Test Smells

Software Refactoring

Software Maintenance & Evolution

Empirical Software Engineering

 https://twitter.com/ShehanPeruma

https://www.peruma.me

Agenda

- Introduction
  - What are identifiers and why are their names important?
- Linguistic Anti-Patterns
  - Introduction to the types of identifier naming violations
- Grammar Patterns
  - Common semantic structures for identifier names
- Tools
  - Some tools that can help developers and researchers with identifier naming
- Conclusion
  - Summary and additional resources

Introduction

Software Maintenance

- Consumes **60% - 80%** of organization resources [1,2]
- Poor maintenance → Low quality software
- Includes:
  - Fixing bugs
  - Incorporating new or updating features
  - Improving the internal quality of the system

[1] Lientz, B. P., Swanson, E. B., & Tompkins, G. E. (1978). Characteristics of application software maintenance. Communications of the ACM, 21(6), 466-471.
[2] R.S. Pressman. Software engineering: a practitioner's approach. McGraw-Hill higher education. McGraw-Hill Education, 2010.

Program Comprehension

- Developers need to understand the code before applying changes or debugging
- 58% of developers time is spent on comprehension activities [1]
- Poor code readability impacts time and quality
- Application growth →
    - More classes/files →
        - More lines of code

[1] Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2017). Measuring program comprehension: A large-scale field study with professionals. IEEE Transactions on Software Engineering, 44(10), 951-976.

Identifiers

- Lexical tokens that uniquely identify elements in the source code
  - Classes, Methods, etc.
- Everywhere in source code – significant part in code comprehension
  - Account for **70% of characters** in the code base [1]
- Must be read to understand behaviour and before any other coding activity
- Automated techniques user identifier data

[1] Deissenboeck, F., & Pizka, M. (2006). Concise and consistent naming. Software Quality Journal, 14(3), 261-282.

Lexical tokens that uniquely identify entities

class name

attribute name

variable name

method name

parameter name

```java
public class ResultsWriter {

    private String outputFile;
    private FileWriter writer;

    public ResultsWriter() throws IOException {
        String time =  String.valueOf(Calendar.getInstance().getTimeInMillis());
        outputFile = MessageFormat.format("Output_{0}.csv", time);
        writer = new FileWriter(outputFile,false);
    }

    public void writeOutput(List<String> dataValues) throws IOException {
        writer = new FileWriter(outputFile,true);

        int i;
        for (i=0; i<dataValues.size(); i++) {
            writer.append(String.valueOf(dataValues.get(i)));

            if(i!=dataValues.size()-1)
                writer.append(",");
            else
                writer.append(System.lineSeparator());

        }
        writer.flush();
        writer.close();
    }
}
```

Responsible for saving/writing results of an operation

Responsible for writing the output which comes in as a parameter

What is a good name?

Crafting names can be challenging – probability of two developer picking the same name is 7% [1]

A strong/high-quality name must reflect its intended behavior

Name should concisely summarize the role of its correlating entity

Good names are important – Hence, many organizations emphasis the use of best practices and coding standards by development teams

High quality identifiers **improves comprehension time by 19+%** [2]

Low quality names can lead to bugs and poor code quality (i.e., more-complex, less-readable and less-maintainable) [3]

[1] Feitelson, D., Mizrahi, A., Noy, N., Shabat, A. B., Eliyahu, O., & Sheffer, R. (2020). How developers choose names. IEEE Transactions on Software Engineering.
[2] Hofmeister, J., Siegmund, J., & Holt, D. V. (2017, February). Shorter identifier names take longer to comprehend. In 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER) (pp. 217-227). IEEE.
[3] Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2010, March). Exploring the influence of identifier names on code quality: An empirical study. In 2010 14th European Conference on Software Maintenance and Reengineering (pp. 156-165). IEEE.

Length matters

Avoid abbreviations and acronyms

Shorter identifiers are more difficult for developers to comprehend [1]

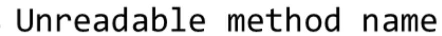
Expanding abbreviations and acronyms is not a straightforward task

- What does num expand to?
    - number
- What does cfg expand to?
    - control flow graph? configuration? configure?

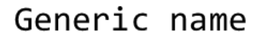
Experienced developers tend to use longer names composed of more words [2]

Longer names are typically composed of 3 or more words [2]

[1] Hofmeister, J., Siegmund, J., & Holt, D. V. (2017, February). Shorter identifier names take longer to comprehend. In 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER) (pp. 217-227). IEEE.
[2] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu and R. Sheffer, "How Developers Choose Names," in IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 37-52, 1 Jan. 2022, doi: 10.1109/TSE.2020.2976920.




```
@Override
protected boolean func_22246_a(int var1)
{
    return false;
}
```



... others are not so straightforward!

Readable attribute name

```
private HashSet includeCrawlingURL;
```

Anti-Pattern: Collection data type, singular identifier name

Readable method name

Anti-Pattern: method name suggests transformation, but no return type

```
public void toSAX(ContentHandler ch) throws SAXException {
    XMLByteStreamInterpreter deserializer = new XMLByteStreamInterpreter()
        ;
    deserializer.setContentHandler(new EmbeddedXMLPipe(ch));
    deserializer.deserialize(this.xmlBytes);
}
```

Challenges with determining the quality of names

- A readable name does not always mean its a high quality name
  - context is important!
- Words are diverse and subjective; for example, a single word can have multiple meanings (homonyms)
  - *Example: Bank can mean rivier bank or financial institution*
- In English prose, the context is provided in natural language, this is not the case with identifier names – context is part of the behaviour of the code
  - *Example: the method: “doForward()” can either refer to an HTTP redirect operation or to move an image on screen*
- Challenge: understanding how to map the meaning of natural language phrases to the behavior of the code

IDENTIFIER NAMING IS HARD

CODING STANDARDS & STYLE GUIDES

Provide heuristics about the overall readability of a class. They do not produce strong names, nor can they provide lexical structure recommendations.

RENAMING

Renames account to over **40%** of the rework developers perform. Renames **do not guarantee a strong name**.

Challenges with renames

A “*rename chain*” - multiple instances of developers renaming an identifier

```
37 -  public void writeMessage( String message )
38    {
39        systemOut.println( message );
40    }
```

```
37 +  public void info( String message )
38    {
39        systemOut.println( message );
40    }
```

```
37 -  public void info( String message )
38    {
39        systemOut.println( message );
40    }
```

```
37 +  public void println( String message )
38    {
39        systemOut.println( message );
40    }
```

Is the final name high-quality?

IDENTIFIER NAMING IS HARD

CODING STANDARDS & STYLE GUIDES

Provide **heuristics** about the overall readability of a class. They do not produce strong names, nor can they provide lexical structure recommendations.

RENAMING

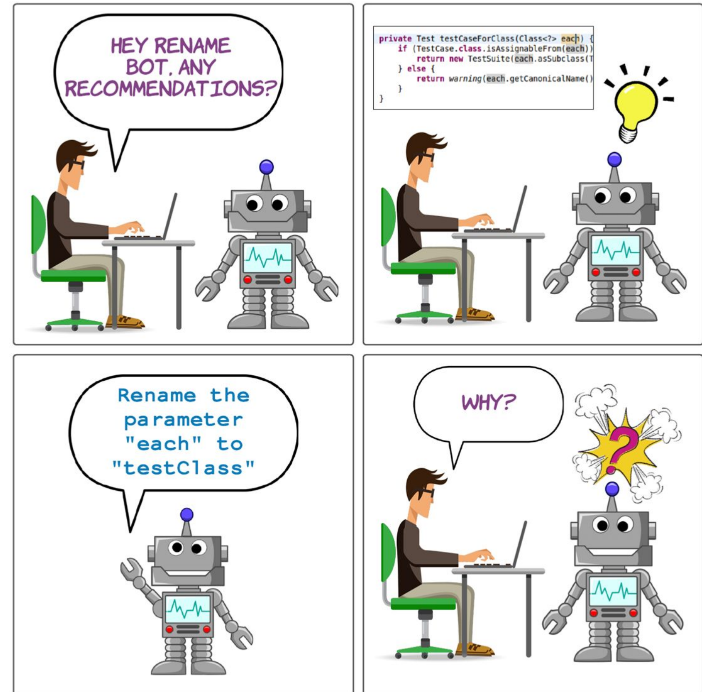
Renames account to over **40%** of the rework developers perform. Renames **do not guarantee a strong name**.

NAME RECOMMENDATION MODELS

Models are **prescriptive not descriptive**. Model is built based on the **existing code styles** and does not consider **pre-existing poor identifiers.** Works **only on method names**. **Context sensitivity** is a challenge.

Challenges with renaming models

- Models only provide name recommendations
- They do not provide details as to why the proposed name is a good replacement
- Does not indicate what parts of the code are influencing the model's recommendation
- Developer will continually make the same naming mistake



Challenges with renaming models

Source code can differ between different environments; a model built and evaluated in one environment will perform badly in another

```java
public String sayHello() {
        Calendar current = Calendar.getInstance();
        if (current.get(Calendar.HOUR_OF_DAY) < 12) {
                return "Good Morning!";
        } else {
                return "Good Afternoon!";
        }
}
```

Production Method

```java
@Test
public void shouldSayGoodMorningInTheMorning() {
        Calendar now = Calendar.getInstance();
        now.set(Calendar.HOUR_OF_DAY, 9);
        DateTimeUtils.setCurrentMillisFixed(now.getTimeInMillis());
        assertEquals("Good Morning!", hello.sayHello());
}
```

Unit Test Method

IDENTIFIER NAMING IS HARD

CODING STANDARDS & STYLE GUIDES

Provide **heuristics** about the overall readability of a class. They do not produce strong names, nor can they provide lexical structure recommendations.

RENAMING

Renames account to over **40%** of the rework developers perform. Renames **do not guarantee a strong name**.

NAME RECOMMENDATION MODELS

Models are **prescriptive not descriptive**. Model is built based on the **existing code styles** and does not consider **pre-existing poor identifiers.** Works **only on method names**. **Context sensitivity** is a challenge.

NLP TECHNOLOGY

Current technology is **built for English prose**– not source code (e.g., Stanford POS tagger); **domain/technology terms** pose a challenge.

Names are diverse, and so are the developers who craft these names -- a one-stop solution is very challenging!

Linguistic Anti-Patterns

Smells

- Smells are specific structures in the code that deviate from fundamental programming practices
- Smells make code harder to understand and make it more prone to bugs and changes
- Smells are a surface indication that usually corresponds to a deeper problem in the software system
- Types of smells:
    - Code Smells (e.g., Long Method, Large Class, Dead Code, etc.)
    - Test Smells (e.g., Assertion Roulette, Eager Test, Lazy Test, etc.)
    - Database Smells (e.g., Multi-purpose column, Tables with many columns, etc.)
    - Linguistic Smells
    - ….

Represent deviations from well-established lexical naming practices in source code

Act as indicators of poor naming quality

Typically take the form of an identifier name that incorrectly describes the behavior of the entity that it represents OR an entity that betrays the behavior conveyed linguistically by its corresponding identifier

Leads to code misinterpretation by developers increasing cognitive load [1]

First conceptualized by Arnaoudova et al. [2]

Catalog of 15+ anti-patterns

[1] Fakhoury, S., Ma, Y., Arnaoudova, V., & Adesope, O. (2018, May). The effect of poor source code lexicon and readability on developers' cognitive load. In 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC) (pp. 286-28610). IEEE.
[2] Arnaoudova, V., Di Penta, M., Antoniol, G., & Guéhéneuc, Y. G. (2013, March). A new family of software anti-patterns: Linguistic anti-patterns. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 187-196). IEEE.

Categories of linguistic anti-patterns

Methods:

1. Do more than they say
2. Say more than they do
3. Do the opposite than they say
4. The entity contains more than what it says

Attributes:

5. The name says more than the entity contains
6. The name says the opposite than the entity contains

Catalog of linguistic anti-patterns

- “Get” more than accessor
- “Is” returns more than a Boolean
- “Set” method returns
- Expecting but not getting single instance
- Not implemented condition
- Validation method does not confirm
- “Get” method does not return
- Not answered question
- Transform method does not return
- Expecting but not getting a collection
- Method name and return type are opposite
- Method signature and comment are opposite
- Says one but contains many
- Name suggests boolean but type is not
- Says many but contains one
- Attribute name and type are opposite
- Attribute signature and comment are opposite

Arnaoudova, V., Di Penta, M., Antoniol, G., & Guéhéneuc, Y. G. (2013, March). A new family of software anti-patterns: Linguistic anti-patterns. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 187-196). IEEE.

Get more than accessor

A getter that performs actions other than returning the corresponding attribute
Example: method *getImageData* which always returns a *new object*

```
ImageData getImageData(){
  final Point size = this.getSize();
  this.imageData = new ImageData(size.x, size.y, 8);
  return this.imageData;
}
```

How to resolve:

1. The method name should change so that it is not a getter or
2. the implementation should be corrected to conform to standard get-method behavior

Is returns more than a Boolean

The name of a method is a predicate suggesting a true/false value in return. However the return type is not Boolean but rather a more complex type thus allowing a wider range of values without documenting them
Example: method *isValid* with return type *int*

```
public int isValid(){
    final long currentTime = System.currentTimeMillis();
    if (currentTime <= this.expires) {
      // The delay has not passed yet -
      // assuming source is valid.
      return SourceValidity.VALID;
    }
  // The delay has passed, prepare for the next interval.
  this.expires = currentTime + this.delay;
  return this.delegate.isValid();
}
```

How to resolve:

1. The type should be changed to boolean to reflect the function's behavior as a binary predicate.
2. Consider changing the name such that it does not imply a yes/no question and provides some indication of n-ary return values.
3. Carefully document the meaning of each value that can be returned. Thoroughly test each value.

Set method returns

A set method having a return type different than void without proper documentation of the return type/values
Example: method *setBreadth* has a non-*void* return type

```
public Dimension setBreadth(final Dimension target, final int source) {
  if (this.orientation == Orientation.VERTICAL) {
    return new Dimension(source, (int) target.getHeight());
  } else {
    return new Dimension((int) target.getWidth(), source);
  }
}
```

How to resolve:

1. The word set, when used in this manner, has a specific definition in the programming domain. Consider using a different term, such as change.
2. Correct the implementation such that it works like a stereotypical set method (i.e., void return, mutates a class attribute)
3. Carefully document the reasoning behind using set while also returning a value

Expecting but not getting single instance

The name of a method indicates that a single object is returned but the return type is a collection
Example: method *getExpansion*, which ends with a head-noun that is singular, but returns a *List* object

```
/**
  * Returns the expansion state for a tree.
  *
  * @return the expansion state for a tree
*/
public List getExpansion() {
  return this.fExpansion;
}
```

How to resolve:

1. Correct the method name so that it is plural-- *getExpansions()*

Not implemented condition

The comments of a method suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.
Example: method *getChildren* has a comment which indicates there should be a conditional within its body.

```java
/**
* Returns the children of this object. When this object is
* displayed in a tree, the returned objects will be this
* element's children. Returns an empty array if this object
* has no children.
*
* @param object The object to get the children for.
*/
public Object[] getChildren(final Object o) {
  return new Object[0];
}
```

How to resolve:
1. Complete implementation of the method
2. Document (i.e., update the comment) that the method is incomplete and does not implement the behavior indicated in its comment

Validation method does not confirm

A validation method (e.g., name starting with "validate", "check", "ensure") does not confirm the validation, i.e., the method neither provides a return value informing whether the validation was successful, nor documents how to proceed to understand
Example: method *checkCollision* returns *void* despite indicating that it is designed to perform validation

```
public void checkCollision(final String before,
                           final String after) {
  final boolean collision = before != null
      && before.equals(this._shortName) || after != null
      && after.equals(this._shortName);
  if (collision) {
    if (this._longName == null) {
      this._longName = this.getLongName();
    }
    this. _displayName = this._longName;
  }
}
```

How to resolve:
1. Change method to return confirmation (i.e., true or false)
2. Consider changing the name to avoid implication of validation behavior (i.e., avoid terms like check and is)
3. If the previous options are not available then thoroughly document method behavior, consider highlighting irregular validation behavior

Get method does not return

The name suggests that the method returns something (e.g., name starts with "get" or "return") but the return type is void. The documentation should explain where the resulting data is stored and how to obtain it
Example: method *getMethodBodies* has a void return type but its name indicates that it is a getter method

```
protected void getMethodBodies(
  final CompilationUnitDeclaration unit,
  final int place) {
    //[Removed some code for conciseness]
    this.parser.scanner
      .setSourceBuffer(
        unit.compilationResult.compilationUnit
        .getContents());
    if (unit.types != null) {
      for (int i = unit.types.length; --i >= 0;) {
        unit.types[i].parseMethod(this.parser, unit);
      }
    }
}
```

How to resolve:

1. Change method to return correct entity
2. Consider changing the name to avoid the word *get*
3. If the previous options are not available then thoroughly document method behavior, consider highlighting irregular getter behavior

Not answered question

The name of a method is in the form of predicate whereas the return type is not Boolean
Example: method *isValid* with a *void* return type

```
public void isValid(final Object[] selection,
                    final StatusInfo res) {
  // only single selection
  if (selection.length == 1
      && selection[0] instanceof IFile) {
      res.setOK();
  } else {
      res.setError(""); //$NON-NLS-1$
  }
}
```

How to resolve:

1. Change method to return correct entity
2. Consider changing the name to avoid the word get
3. If the previous options are not available then thoroughly document method behavior, consider highlighting irregular getter behavior

Transform method does not return

The name of a method suggests the transformation of an object but there is no return value and it is not clear from the documentation where the result is stored.
Example: method *javaToNative* has a *void* return type but indicates that it performs a transformation (i.e., type conversion).

```
public void javaToNative(final Object object,
                         final TransferData transferData) {
  final byte[] check =
      LocalSelectionTransfer.TYPE_NAME.getBytes();
  super.javaToNative(check, transferData);
}
```

How to resolve:

1. Change method to return correct entity
2. If the previous option is not available then thoroughly document method behavior, consider highlighting irregular transformation behavior

Expecting but not getting a collection

The name of a method suggests that a collection should be returned but a single object or nothing is returned
Example: method *getStats* with a *Boolean* return type; making it difficult to understand the reason behind the plurality of the method name.

```java
public boolean getStats() {
  return SAXParserBase._stats;
}
```

How to resolve:

1. Change the name of the method (and any related identifier names) so that it is singular instead of plural

Method name and return type are opposite

The intent of the method suggested by its name is in contradiction with what it returns
Example: method disable with return type *ControlEnableState*. The words "disable" and "enable" having opposite meanings.

```
public static ControlEnableState disable(Control w) {
  return new ControlEnableState(w);
}
```

How to resolve:

1. Change method name so that it aligns better with the return type (i.e., change disable to enable)
2. Change type name to align better with method name (i.e., to ControlDisableState)

Method signature and comment are opposite

The documentation of a method is in contradiction with its declaration
Example: method *isNavigateForwardEnabled* is in contradiction with its comment documenting "a back navigation", as "forward" and "back" are antonyms

```
/**
*       Returns true if this listener has a target for a
*       back navigation. Only one listener needs to return
*       true for the back button to be enabled.
*/
public boolean isNavigateForwardEnabled() {
  boolean enabled = false;
  if (this._isForwardEnabled == 1) {
    enabled = true;
  } else {
    if (this._isForwardEnabled != 0) { enabled =
      this.navigateForward(false) != null;
    }
  }
  return enabled;
}
```

How to resolve:

1. Change the comment to specify that this method is for forward navigation

Says one but contains many

The name of an attribute suggests a single instance, while its type suggests that the attribute stores a collection of objects
Example: *attribute _target* that is of type *Vector*. It is unclear whether a change aspects one or multiple instances in the collection.

```
Vector _target;
```

How to resolve:

1. Change the identifier name to reflect plurality of its type (i.e., _target -> _targets)

Name suggests boolean but type is not

The name of an attribute suggests that its value is true or false, but it's declaring type is not Boolean
Example: attribute *isReached* that is of type *int[]* where the declared type and values are not documented.

```
int[] isReached;
```

How to resolve:

1. Change the name of the identifier to be more descriptive with respect to what kind of array it represents.
2. Consider removing the word is and using a different term unless the array represents a sequence of appropriate (i.e., boolean-like) values
3. If appropriate, consider using a boolean array
4. Carefully document the data represented by the array, including the reasoning for its integer type and whether different integer values have different meanings

Says many but contains one

The name of an attribute suggests multiple instances, but its type suggests a single one
Example: attribute _*stats* that is of type *Boolean*. Documenting such inconsistencies avoids additional comprehension effort to understand the purpose of the attribute.

```
private static boolean _stats = true;
```

How to resolve:

1. Change identifier name to singular instead of plural

Attribute name and type are opposite

The name of an attribute is in contradiction with its type as they contain antonyms
Example: attribute *start* that is of type *MAssociationEnd*. The use of antonyms can induce wrong assumptions.

```
MAssociationEnd start = null;
```

How to resolve:

1. Change identifier name to align with type name (i.e., change start to end)

Attribute signature and comment are opposite

The declaration of an attribute is in contradiction with its documentation
Example: attribute *INCLUDE_NAME_DEFAULT* whose comment documents an "exclude pattern". Whether the pattern is included or excluded is thus unclear

```
/**
*       Configuration default exclude pattern,
*       ie .*\/@href|.*\/@action|frame/@src
*/
public final static String INCLUDE_NAME_DEFAULT
  = ".*/@href=|.*/@action=|frame/@src=";
```

How to resolve:

1. Change identifier name to align with comment (i.e., include -> exclude)
2. Change comment to align with method name (i.e., exclude -> include)

Grammar Patterns

Challenges with determining the quality of names

One challenge to studying identifiers is the difficulty in understanding how to map the meaning of natural language phrases to the behavior of the code.

A second challenge lies in the natural language analysis techniques themselves, many of which are not trained to be applied to software

Part-of-Speech

Part-of-speech is a category to which a word is assigned in accordance with its syntactic functions

In English, the main parts of speech are noun, pronoun, adjective, determiner, verb, adverb, preposition, conjunction, and interjection

Can help us reason about the meaning of words and code behavior

Grammar Patterns

Identifier Phase Structure != Human Language Phrase Structure

A grammar pattern is the sequence of part-of-speech tags assigned to individual words within an identifier

Grammar patterns allow a more efficient analysis by broadly categorizing words into their corresponding part-of-speech

Sample grammar patterns

Noun Modifier (NM) | Noun Modifier (NM) | Noun (N)

```
int dynamic_Table_Index;
```

Verb (V) | Preposition (P) | Noun Modifier (NM) | Noun (N)

```
void save_As_Quadratic_Png();
```

Common identifier naming patterns

Noun Phrase

Zero or more noun-modifiers appear to the left of a head-noun

int *dynamic_Table_Index*;

NM NM N

Plural Noun Phrase

Identical to Noun Phrase, except the head-noun is plural

String[] *method_Name_Prefixes*;

NM NM NPL

Verb Pattern

One or more verbs with no noun phrase

void sort();

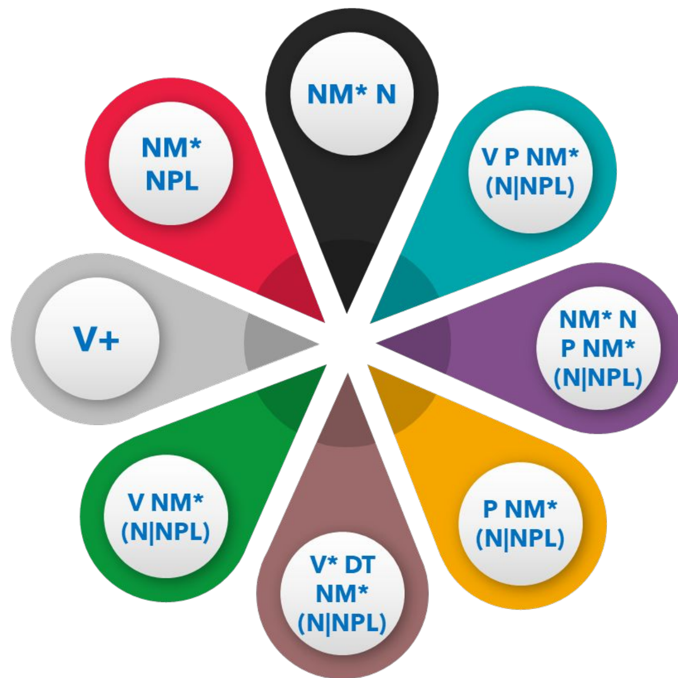
V

Verb Phrase

The addition of a verb to a noun phrase creates a verb phrase

bool create_metadata_array();

V NM N



Prepositional w/ Verb

Prepositional phrase with leading verb

string convert_to_php_namespace();

V P NM N

Prepositional w/ Noun

Prepositional phrase with leading noun phrase

long query_Timeout_In_Milliseconds;

NM N P NPL

Prepositonal Phrase

A noun or verb-phrase with a leading preposition

String to_string();

P N

Noun w/ Determiner

Noun phrase with leading determiner

String[] all_Open_Indices;

DT NM NPL

Common identifier naming patterns

Noun Phrase - common naming pattern for identifiers that are not function names; A good identifier will include only enough noun-modifiers to concisely define the concept represented by the head-noun

Plural noun phrase - Identifiers that follow this pattern are usually not function names; these identifiers are more likely to have a collection data type

Verb Phrase - typically either function identifiers or identifiers with a boolean type; for non-boolean the verb is an action, otherwise it's a predicate

Prepositional Phrase - used in many types of identifiers; The preposition typically explains how the entity represented by the accompanying noun or verb-phrase are related

Noun phrase with leading determiner - used in many types of identifiers; determiner tells us how much of the population, which is specified by the noun-phrase, is represented, or acted on, by the identifier

Verb Pattern -   typically function names or identifiers with a boolean type; missing a noun phrase; the noun phrase is implied by the program context or it is present in the function parameters.

Tools

Tools to analyze/transform identifiers

Naming Violation Detection

- Detects 19 types of linguistic anti-patterns
- Provides an explanation of the violation
- Analyzes C# & Java source code
- Supports project-specific customizations
- Average precision: 75.27%
- Open-source
- https://github.com/SCANL/ProjectSunshine/blob/master/documentaion/IDEAL/SetupAndUse.md

Ensemble Part-of-Speech Tagger

- Tagger uses machine-learning and the output from multiple part-of-speech taggers to annotate natural language text
- The ensemble uses three state-of-the-art part-of-speech taggers: SWUM, POSSE, and Stanford
- Accuracy of 86%; Outperforms Stanford by 51%
- Open-source

Peruma, A., Arnaoudova, V., & Newman, C. D. (2021, September). Ideal: An open-source identifier name appraisal tool. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 599-603). IEEE.

Newman, C. D., Decker, M. J., Alsuhaibani, R., Peruma, A., Mkaouer, M., Mohapatra, S., ... & Hill, E. (2021). An ensemble approach for annotating source code identifiers with part-of-speech tags. IEEE Transactions on Software Engineering.

These are just *some* of the identifier related tools that are available for the developer and research community

Rename recommendation models
G. Li et al., “A Survey on Renamings of Software Entities”, in ACM Comput. Surv.

Code readability models
S. Scalabrino et al., “A comprehensive model for code readability.” in Journal of Software: Evolution and Process

LAPD: linguistic anti-pattern detector
V. Arnaoudova, et al., "Linguistic antipatterns: What they are and how developers perceive them," in Empirical Software Engineering.

Spiral: splitters for identifiers in source code files
M. Hucka, “Spiral: splitters for identifiers in source code files,” in Journal of Open Source Software.

Nominal: Java library to test compliance of identifier names with naming conventions
S. Butler et al., "Investigating naming convention adherence in Java references," 2015 IEEE International Conference on Software Maintenance and Evolution.

Demo

- The Ensemble Tagger
- IDEAL

Conclusion

Summary

- Naming identifiers is one of the most challenging tasks for developers
- A high-quality name should reflect its intended behavior
- Names are diverse and subjective – this makes it challenging to automatically determine their quality
- Linguistic anti-patterns – deviations from lexical naming practices
- Grammar patterns – allow a more efficient analysis of names
- Availability of tools to assist developers with crafting and maintaining names, but they are not a complete one-stop solution

Additional sources

- Identifier Naming Structure Catalogue
  - https://github.com/SCANL/identifier_name_structure_catalogue

Exercise

Can you spot the poor-quality identifier names in this code snippet:
https://replit.com/@peruma/IdentiferNamingViolations?v=1

How would you correct these naming violations?

IdentiferNamingViolations - Java
replit.com/@peruma/IdentiferNamingViolations?v=1#Students.java

replit | Features | Blog | Pricing | Teams Pro | Teams for Education | Careers

IdentiferNamingViolations

Hide files | 0 | Run

Files
- Main.class
- Main.java
- Students.class
- Students.java

Packager files
- target
- pom.xml

```java
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Students {

    private final String name;
    private final int isFullTime;
    private final String major;
    private ArrayList<Double> examGrade;

    public Students(String name, String major, int isFullTime) {
        this.name = name;
        this.major = major;
        this.isFullTime = isFullTime;
    }

    public String getName() {
        return name;
    }

```

peruma + Follow | Made with Java

Thanks!

Anthony Peruma

https://www.peruma.me