Concise and Consistent Naming

Florian Deißenböck and Markus Pizka*
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
{deissenb|pizka}@in.tum.de

Abstract

Approximately 70% of the source code of a software system consists of identifiers. Hence, the names chosen as identifiers are of paramount importance for the readability of computer programs and therewith their comprehensibility. However, virtually every programming language allows programmers to use almost arbitrary sequences of characters as identifiers which far too often results in more or less meaningless or even misleading naming. Coding style guides address this problem but are usually limited to general and hard to enforce rules like "identifiers should be self-describing". This paper renders adequate identifier naming far more precisely. A formal model, based on bijective mappings between concepts and names, provides a solid foundation for the definition of precise rules for concise and consistent naming. The enforcement of these rules is supported by a tool that incrementally builds and maintains a complete identifier dictionary while the system is being developed. The identifier dictionary explains the language used in the software system, aids in consistent naming, and improves productivity of programmers by proposing suitable names depending on the current context.

1. Naming and Comprehension

"The limits of my language mean the limits of my world." — *Ludwig Wittgenstein (1889 – 1951)*

The names of the identifiers used in a computer program resemble the common language of the development team. Referring to philosopher Wittgenstein[1] this language reflects the limits of the common understanding of software systems. Different names used for the same concept or even identical names used for different concepts reflect misunderstandings and foster further misconceptions. Meaningless identifier names as in `class ABZ` are either hints to or a frequent source of a lack of understanding.

We argue that the improvement of identifier naming is a promising opportunity to significantly facilitate program comprehension and as a consequence increase the productivity and quality during software maintenance [16] and evolution [13]. As we will show below, significant improvements can be achieved with moderate costs.

1.1. Evidence

Table 1. Token analysis for Eclipse 3.0M7

Type	#	%	*characters*	%
Keyword	967,665	0.11	4,650,273	0.13
Delimiter	4,096,112	0.467	4,096,112	0.115
Operator	531,444	0.061	669,932	0.019
Identifier	2,873,232	**0.328**	25,646,263	**0.717**
Literal	301,081	0.034	708,308	0.020
Total	8,769,534	1.0	35,770,888	1.0

One does not have to deeply dig into philosophy to understand the importance of identifier naming for program comprehension. Table 1 shows the results of a lexical analysis of the extensive Eclipse[2] (version 3.0M7) Java code base with a total of 2 MLOC[3]. Counting all source level tokens and differentiating according to the type of the token, e. g. keyword or identifier, reveals that 33% of all tokens are identifiers. Since each identifier token consists in average of 8,9 characters, identifiers account for more than two thirds or 72% of the source code in terms of characters. Without further analysis of the complex processes involved in program comprehension it becomes evident that identifiers make up for the bulk of information that a future reader or maintainer of the program has to understand. Though program comprehension is not limited to program reading it

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project VSEK (www.software-kompetenz.de).

[1]http://www.philosophypages.com/ph/witt.htm

[2]`http://www.eclipse.org`

[3]million lines of code

is to expect that the naming of identifiers has a enormous impact on the comprehensibility of a software system.

Another evidence for the strong influence of identifier naming on the comprehension process is one of the strategies commonly applied by code obfuscators: In order to protect the code they substitute the identifiers by meaningless character sequences *(Identifier Scrambling)*[14]. This sole transformation is sufficient to make comprehension a cumbersome task.

```
function mr_mr_1(mr, mr_1)
    if Null(mr) or Null(mr_1) then
        exit function
    end if
    mr_mr_1 = (mr - mr_1)
end function
```

Figure 1. Unwanted obfuscation

Unfortunately, the naming of identifiers in real-world software systems comes often close to obfuscation. Figure 1 shows an example taken from a commercial software system. Though the function itself is not complicated its name does not give any clue what the purpose of the function is. Note that an inline comment describing the function could only provide a marginal improvement because it would not improve the comprehensibility of functions using `mr_mr_1`. Lousy naming in one place spoils comprehension in numerous other places.

1.2. Reasons for Poor Naming

Amongst others, there are three important reasons for the inappropriate naming of identifiers encountered in numerous code bases:

1. Identifiers can be arbitrarily chosen by developers and elude automated analysis.
2. Developers have only limited knowledge about the names already used somewhere in the system.
3. Identifiers are subject to decay during system evolution. The concepts they refer to are altered or abandoned without properly adapting the names. One reason for this is the lack of tool support for globally renaming sets of identifiers referring to the same concept.

Due to 3 naming deficiencies can not solely be explained as a result of neglect by careless programmers. It is indeed practically impossible to preserve globally consistent naming during long-term maintenance and evolution without additional tools. In contrast to a rather simple *rename* refactoring all identifiers names referring to the changed concept must be found and renamed consistently.

1.3. Proposed Solution

The importance of identifier names is neither new nor surprising. At the same time, it is amazing that there is hardly any work that directly deals with identifiers. While forward engineering methods generally tend to ignore long term maintainability issues, re-engineering methods seem to have accepted that identifiers are a weak source of information. Of course, naming conventions are part of countless coding styles [1], [11]. The trouble is, they usually focus on syntactical aspects, e. g.[4]:

- packages: lowercase
- classes: CapitalizedWithInternalWordsToo

When it comes to the actually important aspects of naming, that is the semantics of the names, there is usually little guidance[5]:

> Names should be meaningful in the application domain, not the implementation domain. This makes your code clearer to a reader ...

Clearly, only requiring "meaningful", "descriptive", or "self-documenting" names is insufficient. First, a name not only needs to be meaningful but reflect the correct meaning. Second, the "correct" meaning and name of a concept is naturally highly debatable.

The improvement proposed in this paper aims at filling this gap by rendering the term "meaningful" more precisely. Based on a formal analysis of the properties of identifiers, names, concepts and code and their interrelationships we derive rules for consistent and concise naming of identifiers and provide a concept as well as a tool that helps to enforce the aforementioned attributes of identifiers throughout the system's lifetime.

It is to emphasize that this approach concentrates on comprehensibility rather than analyzability. Clearly, "correct" naming of identifiers can not be checked automatically since it is a semantic property. But like other inherently manual but highly successful quality management techniques like reviews and inspections [10] point out, the inability of complete automation is no argument against a method providing a significant improvement. We follow this route by providing a semi-automatic solution to the naming problem that relies on both manual work and tool-support.

Outline Section 2 discusses related work in the fields of program comprehension and psychology as well as the use

[4] `http://gee.cs.oswego.edu/dl/html/javaCodingStd.html`

[5] `http://www.jetcafe.org/~jim/c-style.html`

of dictionaries in different contexts. In section 3 we pinpoint the problems that usually affect the naming of identifiers based on a formal model of concepts and name spaces and relate these findings to practical experiences in section 4. Section 5 explains our solution to the naming problem on a conceptual level before and outlines the implementation of the concept and the features as well as the resulting tool. Finally, section 6 summarizes our findings and gives a glimpse on future work.

2. Related Work

Program Comprehension Usually, the impact of identifier naming on real-life maintenance activities remains underestimated. Typically, naming rules do not go much further than code formatting guidelines [15] or aren't treated at all even in the context of code formatting and documenting [5]. In [20], Sneed states that in many systems, "*procedures and data are named arbitrarily*".

But, some work discusses the role of identifiers for program comprehension in greater detail. For example, Biggerstaff regards them as hints for the construction of mental representations [7]. In [4] it is stated that "*being able to rely on the names of software artifacts to detect different implementations of the same concept would be very useful*". The naming convention proposed to achieve reliable naming requires amongst others that two software artifacts with the same name should implement the same concept. We broadly agree with these findings but extend them with more precise formal criteria and appropriate tool support.

Raijlich and Wilde [17] mention identifier-based concept recognition as one possible strategy for *concept location*. Concept location is the problem of finding already known concepts in source code which is frequently necessary in maintenance tasks. They state that concept recognition based on identifier names is the most intuitive strategy but argue that it's too fragile due to the dependence on naming skills of the original programmers and loss of meaning during software evolution. They assume that developers only apply more complex strategies like the *dynamic search* [24] method or control and data flow analysis [8] if simpler strategies fail. Shneiderman [19] found that the more complex programs are the more comprehension is aided by meaningful names.

Psychology Research on the cognitive processes of language and text understanding also shows that it is the semantics inherent to words that determine the comprehension process besides syntactic rules. There is clear evidence that the semantic contents of words is processed even *before* the syntactical structure of the sentence is taken into account [3]. Experiments about comprehension processes for ambiguous sentences caused by the presence of homonyms show that readers encountering homonyms are slowed down by the process of mentally activating the different possible meanings of words [3].

Further hints to the importance of naming can be found in psychology, especially in the "Broken Windows" theory [25]. This theory is based on an experiment carried out by Zimbardo and is known to affect software product quality [12]. It stems from the field of crime prevention and proved that a car that already has one window smashed is far more prone to be vandalized than an intact car. It's relevancy for the field of software engineering has long been recognized but is very rarely considered during evolution of long-lived systems. Concerning identifier naming it tells us that there is a high risk of rapid decay once the quality of naming has started to deteriorate.

Weinberg's work on "egoless programming" [23] can easily be extended to the problem of identifier naming since the central objective of egoless programming is "*making the program clear and understandable to the person or people who would ultimately have to read it*".

Dictionaries The use of dictionaries as a means to establish a common understanding of terms has already proved its benefits in some software related fields. Literature on software project management recommends the usage of a *project glossary* or *dictionary* that contains all a description of all terms used in a project. This glossary serves as reference for project participants over the entire project life cycle. An example is the Volere Requirements Specification Method [18] that suggests to use such a dictionary and furthermore advices to "*Select names carefully to avoid giving a different, unintended meaning*".

In database systems it is common practice to maintain a *Data Dictionary* or *Data Directory* [2]. Data dictionaries serve a similar purpose as project glossaries but a more precisely defined in terms of their content and usage. A data dictionary contains the names of all attributes of all tables used in a particular system. For each attribute it stores its data type, a specification of the domain, and a prose description of its meaning. This information serves database managers, programmers, and even users as a valuable foundation for a common understanding of the system.

3. Naming Troubles

Having motivated the importance of names and discussed the extensive work on names and dictionaries we now want to come to a more precise definition of "good" naming practices. We therefore perform a detailed and formal analysis of the properties of names and naming troubles.

3.1. Conciseness by Example

Technically, identifiers are merely syntactic entities acting as aliases for memory addresses where variables, method or classes are stored. But, since the introduction of symbolic names identifiers additionally have to fulfill a far more important purpose that is giving the reader of the program a clue to the concept behind these addresses, i. e. its meaning.

Here, a concept doesn't necessarily have to be a concept of the application domain like an account number. It could as well be a technical concept like a stack or sorting algorithm or a part thereof [17].

A reader of a program tries to map the identifiers read to the concepts they may refer to. The more meaningful, i. e. concise, the names are, the more easily can these mapping be established; compare `stack.push()` with `s.p()`.

The function shown in figure 2 illustrates the importance of conciseness. The name `p` doesn't provide any hint to the concept implemented. Because of this, understanding what `p` actually does is extremely difficult although the function body itself is elegant (taken from an undergraduate exam). Note that general identifiers like `p` increase the comprehension effort in two ways. First, they don't assist the developer in finding reasonable boundaries for the mental models he or she is building. Second, they do not allow to determine at the black box level whether the entity at hand is relevant for a given task or not. The reader is forced to also read and understand the details before being able to judge about its relevancy.

```
fct p = (seq m s) seq seq m :
    p1 (<>, <>, s)
fct p1 = (seq m t, seq m l, seq m r ) seq seq m:
    if r == <> then <>
    elif (rest(r) == <>) ∧ (l == <>) then <t ∘ <first(r)>>
        else p1(t ∘ <first(r)>, <>, l ∘ rest(r)) ∘\
            p1(t, l ∘ <first(r)>, rest(r))
        fi
    fi
```

Figure 2. Function p

Now, changing the name of this function to the descriptive term `transformation`, would already satisfy the requirements of most coding conventions although it is still not concise. Surely, the term `transformation` is helpful since it restricts the possible meanings significantly, e. g. a reader would be able to quickly exclude the possibility of console output. However, `transformation` is still a very general term with too many possible meanings. As such it is not concise enough.

Calling the function `permutation` makes it easy to understand its functionality. "Permutation" restricts the kind of "transformation" implemented in this function. The name is now concise enough to quickly guide the developer to the concept. Now, the reader is left with the rather simple task to check whether or how the function body implements the permutation. This task is trivial compared to having to find the concept without any intuition. The key to it is the conciseness of the identifier.

3.2. Formal Model

Surely, there are countless other examples for weak naming practices and just as many explanations for their pros and cons. To come to well-founded naming rules we now take the observations exemplified above one step further and develop a formal definition of *concise* and *consistent* naming.

3.2.1. Named Concepts.

Let C denote the set of all concepts relevant within a certain scope. The scope is determined by a particular computer program, an application domain, or an organization. A concept is a unit with an associated meaning in terms of properties or behavior. Example concepts at a technical level are a single linked list, a stack but also application level concepts such as an abstract bank account.

C inherently evolves over time. It would be unrealistic to assume that every concept needed was known in advance and no unneeded concepts were in C from the start. As we will show later on, a complete concept space C with all possibly needed concepts would not even be desirable because it would dramatically lengthen the names of the identifiers.

In addition to the concept space C we model all possible names as a set denoted by N and regard the assignment of names to concepts as a formal relation $\mathcal{R} \subseteq N \times C$.

While C determines the expressiveness of a language, N and $\mathcal{R}$ together with a given set of grammar rules define the syntactic representation of its words. Clearly, to maximize comprehensibility of words in this language, i. e. the understanding of behavior formulated in the language, one has to chose N and $\mathcal{R}$ so that the language becomes as simple and intuitive to understand as possible.

3.2.2. Rule 1: Consistency.

The first step towards the postulated simplicity is to enforce a proper relation $\mathcal{R}$ between names and concepts.

There are two different kind of inconsistencies that are also known from natural languages: homonyms and synonyms. *Homonyms* are words with more than one meaning, or more precise:

Definition (Homonym) A name $n \in N$ is called a *homonym* iff $|C_n| > 1$ where $C_n = \{c \in C : (n, c) \in \mathcal{R}\}$.

Homonyms are common in natural languages. An example is the word "book" which can refer to the concept of a book that can be read but also to "book" a flight, and various other concepts. Homonyms occur frequently in computer programs, too. An example is the usage of the identifier name `file` for file handles and filenames alike.

In computer programs, homonyms pose an obstacle for program comprehension since the developer has to take all elements of the set C_n into account when spotting an identifier named n. In real life reverse engineering activities homonyms become even more complicated due to fact that the size and elements of C_n are unknown before all possible meanings denoted with n are found. Figure 3a illustrates a relation $\mathcal{R}$ with the homonym n_1 that refers to concepts c_1 and c_2.

Another core naming problem besides homonyms are synonyms, i. e. different words with the same meaning. Correspondingly we define:

Definition (Synonym) Names $s, n \in N$ are *synonyms* iff

$$C_s \cap C_n \neq \emptyset.$$

While synonyms provide for diversity and elegant formulations in poetry they cause tough problems in computer programs. A typical example for a synonym is the usage of the different identifier names `accountNumber` and `number` for the concept of an account number.

In the absence of homonyms the damage of synonyms is limited since independently from the existence of a synonym an identifier name always clearly hints to a single concept. However, synonyms unnecessarily increase the domain of N and the relation $\mathcal{R}$ and therefore raise the learning effort of the language used. Figure 3b shows a relation with the synonyms n_1, n_2 both referring to concept c_1.

In presence of homonyms, synonyms have a very negative impact strongly increasing the comprehension effort because for each identifier named n the developer has to consider all concepts in

$$M_n = \bigcup_{e \in S_n} C_e$$

where S_n is the set of all names synonym to n (including itself). Figure 3c illustrates this problem: On encountering name n_1 which is synonym to n_2 the reader must take concepts c_1 and c_2 into account.

Obviously, the mixture of synonyms and homonyms, which is commonly found in source codes, maximizes confusion and aggravates comprehension efforts enormously.

Note, that synonyms also aggravate the process of finding locations of a certain concept in a computer program because it is not sufficient to find the modules, classes, methods, or variables with a suitable identifier name but all of these elements with all possible synonymous identifications.

To avoid these troubles we define consistent naming as follows (see fig. 3d):

Definition (Consistency) A naming system C, N, and $\mathcal{R}$ is *consistent* iff $\mathcal{R} \subseteq N \times C$ is a *bijective mapping*. We then define

$$\begin{aligned} n: \quad C &\rightarrow N \\ n(c) &= \text{unique name of concept } c. \end{aligned}$$

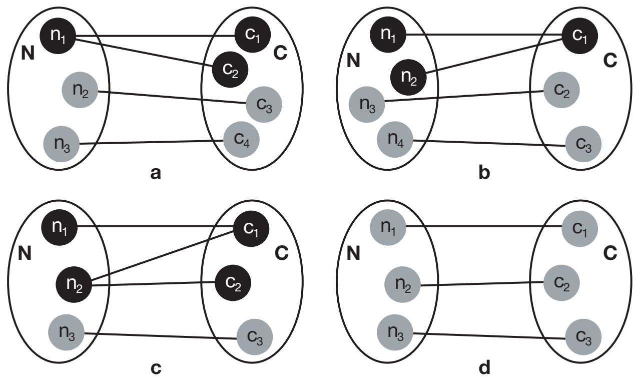


Figure 3. Synonyms and Homonyms

3.2.3. Rule 2: Conciseness. To substantiate *conciseness* we introduce the partial order $\sqsubset$ for the set of concepts C that orders concepts according to the level of abstraction.

For example, it holds that

$$\text{permutation} \sqsubset \text{transformation}$$

because the concept of a transformation is a generalization of permutation; every permutation is also a transformation.

Let the set P contain program elements, such as modules, classes, methods, and variables, that are identified as units via an symbolic name and let i be the mapping of program elements to their identifiers.

$$\begin{aligned} i: \quad P &\rightarrow N \\ i(p) &= \text{identifier of } p. \end{aligned}$$

Let furthermore $[c]$ denote the semantics in the sense of "meaning" of concept $c \in C$. Accordingly $[p]$ denotes the semantics of program element p.

We define *conciseness* in two steps to capture the common problem of identifiers that are either counter intuitive or not concise enough.

First, we require *correct* identification:

Definition (Correctness) Let $p \in P$ be a program element and $c \in C$ the concept it implements, so that $[p] = [c]$. The identifier $i(p)$ for the program element p is *correct* iff the following holds true:

$$i(p) \in \{n(c') : c' \in C \wedge c' \sqsupseteq c\}$$

This means that the identifier of a program element p that manifests concept c must correspond to the name of c or a generalization of c. This rule ensures that identifier names aren't completely meaningless or wrong. So the identifier `p` for the example function in section 3.1 violates the correctness rule because `p` is neither the name of the concept "permutation" nor of a generalization of it. Likewise an incorrect identifier like `load` for this piece of code would be disqualified.

But, correctness alone still qualifies `transformation` as a valid identifier since it corresponds to the name of a generalization of the concept "permutation" though it is of limited help for a reader of the program. This problem is very common: identifiers are somehow correct but not concise enough. To avoid weak identification practices we add the following *conciseness* requirement:

Definition (Conciseness) Let $p \in P$ be a program element and $c \in C$ the concept it implements so that $[p] = [c]$. The identifier $i(p)$ for the program element p is *concise* iff the following holds true:

$$i(p) = n(c)$$

This definition requires an identifier to have exactly the same name as the concept it stands for. It therefore allows the only valid name of `permutation` for the example function.

We now see that identifier naming is completely determined by the characteristics of the set of concepts C. Including all possibly known concepts in set C would require highly sophisticated identifiers such as `completePermutationByCascadedRecursion`. And even this identifier could violate the conciseness criteria if there were more detailed permutation concepts in the concept space that fit the semantics of the program. Hence, besides respecting the above stated rules, the key to keep comprehensibility and detailing of identifiers in balance is to control the content of the concept space C!

4. Experiences

The results of this formal model allow us to precisely explain the identification shortcomings frequently encountered in source code. We illustrate this by discussing experiences made during the development and re-engineering of a sample software project and add results from analyses of Open Source Software Systems.

4.1. Sample Project CloneDetective

The goal of this project was to develop a fast and structured software clone detector [22, 6] tool called *CloneDetective*. Due to rapidly evolving requirements *CloneDetective* underwent various modifications. The development and modification was carried out by the 2 graduate and 10 undergraduate students over a period of one year (graduates) respectively 3 months (undergraduates).

Both, the initial development and the later modifications, were performed without the naming concepts introduced in this paper. Because of this, the observations discussed below are unbiased from certain expectations but reflect typical naming troubles that are circumvented with the naming rules of section 3.2.

4.1.1. Shortcomings During Initial Development. One goal of an enhancement was to make *CloneDetective*'s output more comprehensive by adding information about the position of a clone. Previously, this information was acquired during clone analysis but not stored or presented to the user.

Relevant for this enhancement were especially the two concepts "absolute position" in terms of the complete code base analyzed and a "file-relative position". Unfortunately and probably not uncommon, an analysis of the source code exposed eight (!) different identifiers for these two concepts: `x`, `pos`, `apos`, `abspos`, `relpos`, `absolutePosition`, `relativePosition` and `position`.

This arbitrary and misleading naming proved to highly increase the comprehension effort since the students implementing the new feature could never be sure which kind of position was meant at a particular location and had to go through intensive debugging sessions to fulfill their actual maintenance task of enriching *CloneDetective*'s output.

Though everyone knew, that this naming was troublesome, there was no solid argument what the correct naming would have to be. Now, with the formal model introduced above it is possible to give a detailed explanation of the problems experienced.

The relevant concepts are c_1 = "absolute position", c_2 = "relative position" and the implied more general concept c_3 = "position". The concepts are ordered in the following way.

- $c_1 \sqsubset c_3$ ("position" is more abstract than "absolute position")
- $c_2 \sqsubset c_3$ ("position" is more abstract than "relative position")

Now it becomes evident, that the naming of identifiers in the *CloneDetective* contains the synonyms `apos`, `abspos`, absolutePosition for concept c_3 and relpos, relativePosition for c_2. This clearly violates the consistency rule. $\mathcal{R}$ is obviously not bijective with these identifier names. In fact, it is not even a mapping.

In addition to this synonym defect, identifier x violates the *correctness* rule because it does not match any concept name. And the identifiers named position violate the *conciseness* rule. Identifying a variable storing an absolute or relative position a "position" is *correct* but not *concise*. The reader cannot know whether an absolute or relative position is meant; which in fact proved to be relevant during program analysis and modification.

4.1.2. Decay. Subsequent extensions of the *CloneDetective* delivered an instructive example for the threat of code decay [9] during evolution with respect to identifier naming.

In the beginning a simple line-based detection mechanism was used for clone detection which was later on extended to a more flexible unit-based analysis with units of varying granularity. Before this enhancement, the identifier line was both correct and concise. But switching to a unit-based process demanded identifiers referencing the more abstract unit concept ("line" $\sqsubset$ "unit"). So, by abandoning the line-based concept the existing identifier line not only lost its conciseness but also became *incorrect* because the line-based concept was removed from the set of concepts C.

Even month after this change one could find identifiers named line in almost all modules of the program. Firstly this didn't really pose a serious problem because the identifiers line and unit just became synonyms. But, by accepting this situation inconsistent naming was acquiesced.

Noticeable problems arose when the line-based detection was re-introduced as an optional component. We now had concise identifiers unit that properly referenced the unit concept and line identifiers that also concisely referenced the line concept. But additionally there were now identifiers named line for program elements implementing the unit concept! Thus, the identifier line was always correct, but became a homonym with severe consequences.

Ignoring this problem for a couple of weeks finally gave rise to the effects of the "'Broken Windows"' theory (see 2). Students working on the program were not able to comprehend its original meaning in many places. They generally considered it at mess and started using arbitrary names for both concepts in pretty much every module of the program. As a result further work on the program got more and more complicated and error-prone. The re-engineering effort needed to clean up this mess was extensive and is still not completed although the program is fairly small (13,000 LOC).

4.2. Naming in Open Source Software

Section 1 presented the result of an analysis showing, that approximately 70% of all characters of Eclipse's code base are identifiers. To gather further insights about identifier naming issues we studied the number of different identifiers. In the case of Eclipse 3.0M7 this yields the astonishing figure of 94,829 different identifiers which is around the same number of words as in Oxford Advanced Learner's Dictionary. Naturally this high number stems mostly from usage of compound identifiers like getCounter. The fact that identifiers in Java are commonly written in CamelCase [1] allows the application of a simple heuristic breaking the compounds in distinctive words. Breaking the compounds and counting only different words still resulted in 7,233 different words. Considering the fact that speakers of English as second language need a vocabulary size of around 5,000 words to understand academic texts [21] this figure still seems suspiciously high.

One explanation for this large number is the frequent occurrence of synonyms. In fact a manual inspection of the alphabetically ordered list of all identifiers in Eclipse revealed that almost all identifiers are grouped in blocks with very similar names like: frag, fragement, fragment, fragmentation, fragmented, fragmentname, fragments, frags.

A token analysis of the source code of Sun's JDK 1.4.2 (1.3 MLOC) shown in figure 2 demonstrates that the high percentage of identifiers in Eclipse (fig. 1) is no exception but rather the rule. Again identifiers account for about 30% of tokens or almost 70% of the code in terms of characters. The JDK features 42,869 different identifiers that are compounds of 6,426 different words. Taking JDK's smaller size overall into concern this number appears even higher than the one of Eclipse. Manual inspection of the list of identifiers again reveals a plethora of seemingly synonym words.

Table 2. Token analysis for Sun JDK 1.4.2

Type	#	%	*characters*	%
Keywords	417,274	0.118	2,013,750	0.155
Delimiters	1,609,445	0.457	1,609,445	0.124
Operators	248,619	0.071	314,363	0.024
Identifiers	1,083,508	**0.307**	8,668,782	**0.668**
Literals	166,708	0.047	362,468	0.028
Total	3,525,554	1.0	12,968,808	1.0

Repeating the same analysis for Tomcat 5.0.30 (317 kLOC) confirmed the previous results. As table 3 indicates, identifiers again accounted for about 30% of tokens or almost 70% of the code in terms of characters. Tomcat's source code has 11,656 different identifiers composed from 2,587 words. Our assumption about synonyms is again backed by the results of the manual inspection of identifiers.

Table 3. Token analysis for Tomcat 5.0.30

Type	*#*	*%*	*characters*	*%*
Keywords	105,903	0.109	517,425	0.147
Delimiters	458,113	0.47	458,113	0.13
Operators	59,575	0.061	72,169	0.02
Identifiers	303,687	**0.312**	2,384,250	**0.676**
Literals	47,498	0.049	96,207	0.027
Total	974,776	1.0	3,528,164	1.0

Based on this results it should be evident that identifier naming is a severe problem in real life software systems. All of the shortcomings experienced and found can be both explained and avoided by means of the formal naming rules defined in 3.2.

5. Tool Support: The Identifier Dictionary

Explanations and rules are useful but not enough to effectively improve Software Engineering practices in general and naming practices in particular. Without any further support the controlling of identifier conciseness and consistency would certainly be a tedious and unreliable task.

A promising and time-saving approach to put these formal considerations into effect is to setup and maintain a tool-supported *Identifier Dictionary (IDD)* with each software system. The concept of the IDD is inspired and works similar to a *Data Dictionary*. Basically, it is a database that stores information about all identifiers such as their name, the type of the object being identified and a comprehensive description.

At first glance it seems that an IDD would introduce enormous overhead. In reality all of development, maintenance and quality assurance can benefit from a carefully designed and implemented IDD.

- *Development:* Developers can use the IDD to search for already existing identifiers before creating new ones. This reduces the risk of creating synonyms and helps to choose identifier names that follow existing naming patterns.

- *Maintenance:* The IDD assists comprehension processes by enabling simple and fast lookups of meanings (or at least description) of identifiers. It also helps to locate concepts by providing a list of all relevant concepts and corresponding identifier names. So maintainers are able to browse or search for particular concepts and then locate the corresponding identifiers in the source code.

- *Quality assurance:* The IDD allows to review important aspects of identifiers with moderate effort. It supports conciseness checks by comparing identifier names with their description. Consistency can be reviewed by manual inspections of the identifier list and the associated descriptions and types. Further options are offered by automatically tracking changes in the IDD. For example, a maintenance task resulting in dozens of new identifiers is definitely suspicious and a candidate for manual inspection.

5.1. Requirements

Naturally the manual creation and maintenance of an IDD for a large system must be considered unrealistic. Elaborated tool support is necessary. Basic requirements for an appropriate tool are:

- The core functionality of the IDD tool is the storage of all identifiers in a repository. The tool should furthermore be capable of collecting all identifiers and their type automatically from the source code. Users must be able to enter descriptions of the identifiers and to browse or search the dictionary.

- To maximize developer's comfort the tool should be seamlessly integrated in an Integrated Development Environment (IDE) and thereby allow access to the dictionary without switching between the IDE and other applications.

- The tool should enable the developer to browse source code in a identifier-guided fashion. For example, developers should be able to look up a particular identifier in the source code, ask the tool for a list of all occurrences of the identifier, and navigate to selected occurrences.

- The tool should reduce naming deficiencies and improve productivity by providing an advanced auto-completion feature for identifiers.

- The tool has to provide global *rename* refactorings; e. g. a consistent renaming of all identifiers called `line` to `unit` in the entire program. Currently IDEs only support renaming of single identifiers within the scope they're defined but not all identifiers with a certain name.

- The tool must provide easy access to the dictionary for quality assurance. This could be a database interface or an export feature that creates readable HTML representation of the dictionary.

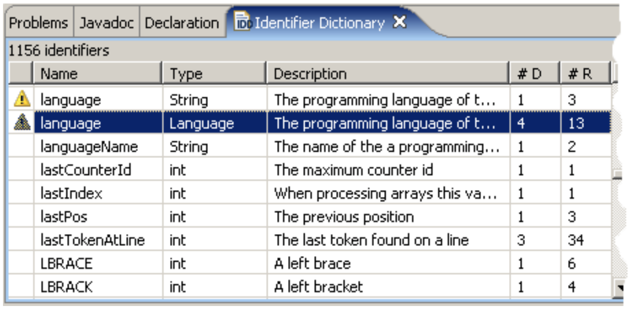
5.2. Implementation

According to these requirements, we implemented an IDD tool as a Plug-In for the Eclipse Java Development

Platform[6]. Eclipse was selected as a basis for the IDD tool because it allows seamless integration of the IDD into the existing IDE and provides broad functionality that helped to keep the development effort low.

Eclipse uses the notion of "projects" to structure the development of software systems consisting of a variety of artifacts. The IDD Plug-In can be enabled or disabled for particular projects. Once the IDD support is turned on an *identifier collector* is installed. This collector is coupled with Eclipse's build process and collects all identifiers while the program is being compiled. It performs an analysis of the abstract syntax tree of each source file to also automatically determine the types associated with the identifiers found. Analysis is done in an incremental fashion so only previously changed source code will be analyzed. Identifier descriptions are either kept in an XML file or a database for team-wide usage.

For instant access to the identifiers and their descriptions the IDD Plug-In contributes a new view to the Eclipse workbench displaying a sorted list of all identifiers in a table (fig. 4).


Problems | Javadoc | Declaration | Identifier Dictionary

1156 identifiers

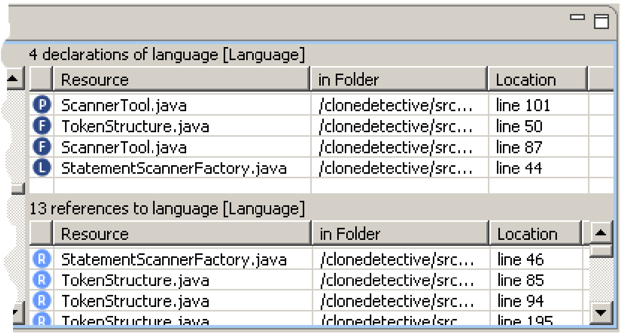
	Name	Type	Description	# D	# R
⚠	language	String	The programming language of t...	1	3
⚠	language	Language	The programming language of t...	4	13
	languageName	String	The name of the a programming...	1	2
	lastCounterId	int	The maximum counter id	1	1
	lastIndex	int	When processing arrays this va...	1	1
	lastPos	int	The previous position	1	3
	lastTokenAtLine	int	The last token found on a line	3	34
	LBRACE	int	A left brace	1	6
	LBRACK	int	A left bracket	1	4

Figure 4. Identifier Dictionary View (left)

All identifiers are listed with their name, type, a prose description, and the number of declarations of identifiers with the same name. Via context menu entries it is possible to open a dialog and edit the description of an identifier. On selecting an identifier the right part of the identifier view (fig. 5) displays two lists of occurrences of the identifier with details on the various declarations of identifiers with the selected name and references to them. The icon in the left-most column indicates the type of declaration like local variable, field or method. The other columns specify the exact location of occurrence of the identifier. Double-clicking on an occurrence opens an editor window an sets the cursor to the specified location.

During the collection process identifiers may become annotated with warnings indicating potential consistency problems. These are shown with warning icons in the IDD view (fig. 4). Currently the IDD features two basic warn-

[6]`http://www.eclipse.org/jdt`


4 declarations of language [Language]

	Resource	in Folder	Location
P	ScannerTool.java	/clonedetective/src...	line 101
F	TokenStructure.java	/clonedetective/src...	line 50
F	ScannerTool.java	/clonedetective/src...	line 87
L	StatementScannerFactory.java	/clonedetective/src...	line 44

13 references to language [Language]

	Resource	in Folder	Location
R	StatementScannerFactory.java	/clonedetective/src...	line 46
R	TokenStructure.java	/clonedetective/src...	line 85
R	TokenStructure.java	/clonedetective/src...	line 94
R	TokenStructure.java	/clonedetective/src	line 195

Figure 5. Identifier Dictionary View (right)

ings: If two identifiers with identical name but different type are found the identifiers are annotated to give a hint to a consistency violations. A typical example is the usage of the identifier `file` for objects of the class `File` and for a `String` for the name of the file. The latter one should rather be called `filename`. Another annotation is used if an identifier is declared but never referenced. This allows an easy detection of superfluous identifiers. Most development environments already have built-in functionality for this kind of analysis but normally limit it to local variables and private class members. The IDD's detection of unreferenced identifiers works for all kinds of identifiers including methods regardless of their visibility.

Further assistance for developers is provided by *hover popups* that offer access to the IDD while editing source code. Placing the mouse cursor over an identifier automatically retrieves the description stored for this identifier from the IDD and displays it at the current position. By using this feature the developer can query the IDD without leaving the editor and switching to another view.

Modern development environments provide extensive auto-completion features for keywords, previously declared variables, typical programming constructs (e. g. switch-statements), and the selection of methods from a given object. They fail to provide auto-completion for identifier names that are not declared in the scope of the current editing location. Hence, it is not possible to complete a variable declaration statement that starts with `int abso...`. The IDD offers the possibility to extend the auto-completion feature for all statements using or declaring identifiers. Given that identifier `absolutePosition` is stored in the IDD the developer can request auto-completion after typing `int abso...` and it will be completed to `int absolutePosition`. If more than one match is found the IDD provides a list of identifiers from which the developer can choose. The matching also takes the type of the identifier (e. g. `int`) into account.

The IDD plug-in features a refactoring called *global rename* that supports consistent global renaming of all identifiers with a certain name. The implementation is built on Eclipse's refactoring capabilities and supports renaming of all kind of identifiers including local variables, fields, types and methods. The global renaming feature provides a detailed preview of all proposed changes to the source code and fully automated validity checking. So the developer does not have to carry out extensive manual renamings that would result in incorrect code to preserve consistency and conciseness of identifier naming while evolving the system.

Quality assurance may directly query the database the dictionary is stored in. Additionally the IDD can be exported as an HTML file. This file provides a clear and easy to read format of the IDD content.

6. Conclusion

While a rich ambiguous language is a quality attribute in poetry, synonyms and homonyms as well as meaningless names are a heavy burden for program comprehension. Proper naming of identifiers is of paramount importance for program readability. We explained what proper naming actually means with the help of a formal model and gave various examples. The resulting consistency and conciseness rules are easy to communicate and allow unbiased checking. The enforcement of this rules is supported with a tool that implements a globally consistent *identifier dictionary* (IDD). The tool not only reduces the effort needed to comply with the naming rules but also provides support for standard tasks such as determining identifier names in declarations.

The identifier dictionary can't constitute the sole remedy for the problem of imprecise and inconsistent naming of identifiers. There also has to be a continuous process to establish and maintain a common understanding of valid terms as well as their meanings among the participants of a software project. So far, there are no direct experiences with the IDD and such a process though there are numerous hints to its benefits from various fields and probably every developer with significant experience will agree that identifier naming is crucial for the readability of code. However we are now highly interested in answering the question how readable code actually can be. It would be interesting to see how "comprehensible" a mid to large scale code base can actually get by strictly complying to our naming rules, consequently using an IDD, and also respecting further code formatting rules. In addition to this our future work will investigate the composition of identifier names. Many real life identifiers are not atomic but composed out of different words. However, there are hardly any well-founded rules for the correct composition of names though there is an obvious difference between a `futureWorkStack` and a `workStackFuture`.

References

[1] Code conventions for the Java programming language. Technical report, Sun Microsystems, Santa Clara, CA, 1999.

[2] F. W. Allen, M. E. S. Loomis, and M. V. Mannino. The integrated dictionary/directory system. *ACM Comput. Surv.*, 14(2):245–286, 1982.

[3] J. R. Anderson. *Cognitive Psychology and its implications*. W. H. Freeman and Co., New Jersey, 1995.

[4] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98*, page 4. IBM Press, 1998.

[5] M. Arab. Enhancing program comprehension: formatting and documenting. *SIGPLAN Not.*, 27(2):37–46, 1992.

[6] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. 1998.

[7] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*. IEEE CS Press, 1993.

[8] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *IWPC '00*, page 241. IEEE CS, 2000.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

[10] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[11] M. Henricson and E. Nyquist. Programming in C++: Rules and recommendations. Technical report, Ellemtel Telecommunication Systems Laboratories, 1992.

[12] A. Hunt and D. Thomas. *The pragmatic programmer: From journeyman to master*. 1999.

[13] M. Lehman. Software evolution threat and challenge. Professorial and Jubilee Lecture, Oct. 2003. 9th international Stevens Award, hosted by ICSM 2003.

[14] D. Low. Protecting Java code via code obfuscation. *Crossroads*, 4(3):21–23, 1998.

[15] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *ACM Communications*, 33(5), 1990.

[16] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[17] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*, page 271. IEEE CS, 2002.

[18] J. Robertson and S. Robertson. Volere template v10.1. Requirements specification template, Atlantic Systems Guild, 2004.

[19] B. Shneiderman. *Software psychology*. Winthrop Publishers, Inc., 1980.

[20] H. M. Sneed. Object-oriented cobol recycling. In *WCRE '96*, page 169. IEEE Computer Society, 1996.

[21] E. Tschirner. Breadth of vocabulary and advanced english study: An empirical investigation. *Electronic Journal of Foreign Language Teaching*, 1(1):27–39, 2004.

[22] F. van Rysselberghe and S. Demeyer. Evaluating clone detection techniques. In *ELISA 03*, pages 25–36, 2003.

[23] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., 1971.

[24] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[25] J. Q. Wilson and G. L. Kelling. Broken windows. *The Atlantic Monthly*, 249(3), 1982.