Software Qual J (2006) 14:261–282
DOI 10.1007/s11219-006-9219-1

Concise and consistent naming*

Florian Deissenboeck · Markus Pizka

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 somehow 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 supports 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

* 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).

F. Deissenboeck (✉) · M. Pizka
Institut für Informatik, Technische Universität München Boltzmannstr. 3, D-85748 Garching b. München, Germany
e-mail: deissenb@in.tum.de
e-mail: pizka@in.tum.de

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

Table 1 Token analysis for Eclipse 3.1.1

Type	*#*	*%*	*characters*	*%*
Keywords	1,321,005	11.2	6,367,677	12.7
Delimiters	5,477,822	46.6	5,477,822	11.0
Operators	701,607	6.0	889,370	1.8
Identifiers	3,886,684	**33.0**	35,723,272	**71.5**
Literals	378,057	3.2	1,520,366	3.0
Total	11,765,175	100.0	49,978,507	100.0

Fig. 1 Unwanted obfuscation

```
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
```

limits of the common understanding of a software system. 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 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 therewith increase the productivity and quality during software maintenance (Pigoski, 1996) and evolution (Lehman, 2003). As we will show below, significant improvements can be achieved with moderate costs.

1.1. Evidence

One does not have to dig deeply 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 3.3.1) Java code base with a total of 2.7 MLOC[3]. Counting all source level tokens and differentiating them according to their kind, e.g. keyword or identifier, reveals that 33% of all tokens are identifiers. Since each identifier token consists in average of 9.2 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 is to expect that the naming of identifiers has an 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)* (Low, 1998). This sole transformation is sufficient to make comprehension a cumbersome task.

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

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

[3] million lines of code

the function is. Note that an inline comment describing the function would only provide a marginal improvement because it would still not improve the comprehensibility of other 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 else 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 development efforts, maintenance, or evolution without additional tools. Note, in contrast to a rather simple *rename* refactoring all identifiers names referring to a concept must be found and renamed consistently each time the concept itself changes.

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 tend to ignore long term maintainability issues anyway, 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 (Sun Microsystems, 1999), (Henricson and Nyquist, 1992). 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,

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

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

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 (Fagan, 1976) 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 of dictionaries in different contexts. In Section 3 we pinpoint a definition for concise and consistent naming of identifiers based on a formal model that relates concepts with name spaces. Section 4 extends the scope of rules on correct identifier naming by regarding name composition before we relate our formal findings to practical experiences in Section 5. Section 6 describes the technical solution to the naming problem on both the conceptual and implementation level including tool-support. Finally, Section 7 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 largely underestimated. Typically, naming rules do not go much further than code formatting guidelines (Oman and Cook, 1990) or are not treated at all even in the context of code formatting and documenting (Arab, 1992). In (Sneed, 1996), 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 (Biggerstaff et al., 1993). In Anquetil and Lethbridge (1996) 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.

Rajlich and Wilde (2002) 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 is 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* (Wilde and Scully, 1995) method or control and data flow analysis (Chen and Rajlich, 2000) if simpler strategies fail. Shneiderman (1980) 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 (Anderson, 1995). 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 (Anderson, 1995).

Further hints to the importance of naming can be found in psychology, especially in the "Broken Windows" theory (Wilson and Kelling, 1982). This theory is based on an experiment carried out by Zimbardo and is known to affect software product quality (Hunt and Thomas, 1999). 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. Its 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" (Weinberg, 1971) 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 various software related fields. Literature on software project management recommends the usage of a *project glossary* or *dictionary* that contains 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 (Robertson, 2004) 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* (Allen et al., 1982). Data dictionaries serve a similar purpose as project glossaries but are more precisely defined in terms of their content and usage. A data dictionary contains the names of all attributes used in a particular database 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 does not 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 (Rajlich and Wilde, 2002).

```
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
```

Fig. 2 Function p

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 easier it is to establish this mapping; compare `stack.push()` with `s.p()`.

The function shown in figure 2 illustrates the importance of conciseness. The name `p` does not 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 unspecific identifier names like `p` increase the comprehension effort in two ways. First, they do not 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.

Now, changing the name of this function to the descriptive term `transformation`, would already satisfy the requirements of most coding conventions, i.e. self-describing and meaningful, although it is still not concise. Surely, the term `transformation` is helpful since it restricts the possible meanings significantly, e.g. a reader could 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 were 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.

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 behavior or properties. Examples for concepts are a single linked list or a stack at the technical level 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.

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 kinds 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 $n, m \in N$ are *synonyms* iff

$$C_n \cap C_m \neq \emptyset.$$

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 every name will always clearly denote a single concept independently from the existence of synonyms. 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 an extremely negative impact that strongly increases the comprehension effort because for each identifier named n the developer has to

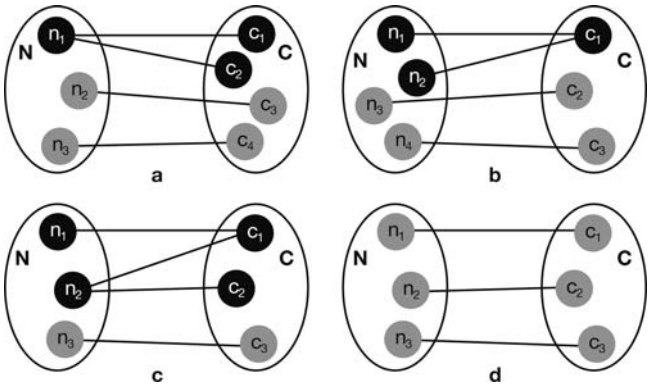

Fig. 3 Synonyms and Homonyms

consider all concepts in

$$M_n = \bigcup_{m \in S_n} C_m$$

where S_n is the set of all synonyms of 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

$$n : \ C \ \rightarrow N$$
$$n(c) = \text{unique name of concept } c.$$

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.

In linguistics this relation between two words is called *hypernymy*: permutation is a *hyponym* (subordinate) of transformation and transformation is a *hypernym* (superordinate) of permutation.

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

$$i: \ P \ \rightarrow N$$
$$i(p) = \text{identifier of } p.$$

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 hypernym of c. This rule ensures that identifier names are not completely meaningless or wrong. The identifier `p` for the example function in Section 3.1 violated this correctness rule because `p` was 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 (if possible) would require highly sophisticated identifiers such as

`completePermutationByCascadedRecursion`.

And even this identifier could violate the conciseness criterion if there were more detailed permutation concepts in the concept space fitting the semantics of the program. Hence,

Table 2 Distribution of compound and non-compound (atomic) identifiers

Program	*# Idents.*	*# Atom. Idents.*	*# Comp. Idents.*	*% Comp. Idents.*
Eclipse 3.0M7	98,947	10,279	88,668	89
SUN JDK 1.5.0	64,652	11,054	53,598	82
Tomcat 5.5.14	14,766	2,897	11,869	80

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. Compound identifiers

Complying with the rules defined in the previous section quickly leads to a new problem: natural languages do not provide enough different terms for all concepts and their variations. It is therefore necessary to create new names through name composition.

Evidence for this can be found by analyzing existing source codes. Regarding typical formatting rules for compound identifiers, e.g. underscores or capitalization, to distinguish between compound and non-compound identifiers one can see that about 80% of all identifiers in large programs are compounds (table 2).

Due to the frequency of compound identifiers precise composition rules are mandatory to reach a truly concise and consistent naming system.

Although there might be an intuitive understanding of the difference between compound and atomic names a precise definition is necessary to explain how our model can be extended to compound names.

*Definition (*Compound names). A name n is a *compound* iff it is the result of a concatenation, i.e.:

$$n = n_1 \circ n_2 \circ \ldots \circ n_m; n_i \in N.$$

Names that are not *compound* are called *atomic* and cannot be split up.

So far this only defines the nature of compound names but does not give instructions on their construction. As name composition is common in natural languages we first investigated composition rules applied there.

4.1. Compound words in english

As English can be considered a lingua franca in computer programs the following focuses on it. However, it can expected that most of the observations can be transferred to other languages with similar properties like English.

Note, that compounds in natural languages do not necessarily have to be written as one consecutive sequence of letters. Orthographical rules regarding this issue are ambiguous in most languages and change over time. An example is the word *long-sighted* which can be written as two distinct words, with a hyphen or without a hyphen as in *longsightedness*. Here, we will not further distinguish these different writings but concentrate on the actual composition taking place and the role of the ordering of words. For example, almost anybody will have an intuitive understanding for the difference between the words *manager-office* and *office-manager*.

In the field of *morphology*, a branch of linguistics, there is extensive research on processes of creating and understanding compound words in natural languages. Unfortunately most of this research concludes that word composition is usually very ambiguous and there are very few strict rules concerning the creation and interpretation of compounds. In English it is in fact not even clear if a group of words actually forms a compound or not. E. g. it is impossible to identify the meaning of the words *green house* in the following sentence without knowing the context: "'Last night someone broke into the green house"'. It could either be a house painted green or a special kind of building for growing plants.

Noun-noun and adjective-noun compounds. In English as well as in programs the most frequent kind of compounds are *noun-noun compounds*. Although most of the research regarding noun-noun-compounds deals with a plethora of interesting (for linguists) special cases and phenomena it is a generally accepted result that most compounds involve a *head concept* and a *modifier concept*. This means that the compound describes a specialization of the head concept; the compound is a hyponym of the head. Compounds of this kind are called *endocentric*. In English the last word of the compound is almost always the head of the compound (O'Grady and de Guzman, V.P, 1996). Examples are:

- *Blackboard*. *Board* is the head concept and *Black* is the modifier: *Blackboard* is a special kind of *board*.
- *Raincoat*. *Coat* is the head concept and *Rain* is the modifier: *Raincoat* is a special kind of *coat*.
- *Handbag*. *Bag* is the head concept and *Hand* is the modifier: *Handbag* is a special kind of *Bag*.

There are, however, other less frequent groups of compounds that do not comply with this rule. Examples are *exocentric* compounds and *copulative* compounds. Exocentric compounds do not have a clearly identifiable head, e.g. *pickpocket, lazy-bones*, while copulative compounds have more than one head, e.g. *fighter-bomber, pantskirt*. Exocentric compounds demand particularly complex comprehension processes as they rely on the comprehension of something other than the elements of the compound itself. For example the exocentric compound *couch potato* remains completely meaningless without appropriate further knowledge.

Verb-noun compounds verb-noun compounds. are rare in English as the verb is usually substituted by a gerund, e.g. *bloodcurdling*. More important are verb-noun compounds like *getCounter* that are typically found in programs but hardly considered compounds in natural languages. From a linguistic perspective they can rather be treated like *imperative sentences*.

4.2. Composition rules

Unfortunately, composition rules used in natural languages are way too loosely defined and too ambiguous to serve as a useful definition of correct name composition in computer programs. There is even evidence that the creation of compounds is stronger influenced by the conceptual representations people have than by the communicative tasks they carry out. That is to say that the creation of compounds is usually *not* targeted at creating compounds than can be understood easily (Costello, 2002).

Since this aspect of name composition in natural languages is inadequate for software systems, it is necessary to set more precise rules for the composition of identifiers in computer programs. The primary goal of these rules are to facilitate understanding and to eliminate ambiguities.

The most basic problem is the identification of compounds. As most programming languages do not allow spaces or hyphens within identifiers, the start and end of a compound identifiers is easy to detect. However it may become difficult and even ambiguous to identify the elements within a compound such as `illegalargumentexception`. Therefore we propose to consistently use a notation that clearly separates the atomic elements of a compound identifier. Possible notations typically rely on underscores, as in `illegal_argument_exception`, or capitalization as in `IllegalArgumentException`.

As the composition based on a head and a modifier concept is the most frequent and most intuitively understood method of composition in natural languages we adapted it for the composition of identifiers:

*Definition (*Valid composition). A composition is valid, iff

$$n(c_3) = \underbrace{n(c_1)}_{\text{modifier}} \circ \underbrace{n(c_2)}_{\text{head}} \rightarrow c_3 \sqsubset c_2$$

The concept identified by a compound identifier must be a specialization of the concept identified by the head of the compound identifier. Note, that this composition scheme can and will often be used in a recursive fashion with more than two atomic elements. A good example is the identifier `LinkedHashMap` that represents a class of Sun's JDK 1.5. The concept *LinkedHashMap* is a specialization of the concept *HashMap* which in turn is a specialization of the concept *Map*.

This example highlights an issue relevant in object-oriented systems: if properly designed and named, the class hierarchy parallels the concept hierarchy with the corresponding name hierarchy.

The composition rule aids program comprehension by giving compound identifiers a precise semantic and suggesting a black-box/white-box separation on a very fine-granular level. Let, e.g., the concept *counter* be defined as an entity that counts other entities and is therefore associated with a positive natural number. On the occurrence of identifiers like `argumentCounter` or `lineCounter` the developer can, in certain contexts, deduce important aspects about the nature of the related concepts by considering only the head concept.

To avoid ambiguities no other compositions schemes besides endocentric ones are allowed. Exocentric and copulative compositions would unnecessarily complicate comprehension efforts and are, as their relative low frequencies in natural languages show, dispensable.

5. Experiences

The results of this formal model allow us to precisely explain the identification shortcomings frequently encountered in source codes. 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.

5.1. Sample project CloneDetective

The goal of this project was to develop a fast and structured software clone detector (van Rysselberghe and Demeyer, 2003; Baxter et al., 1998) tool called *CloneDetective*. Due to rapidly evolving requirements *CloneDetective* underwent various modifications. The

development and modification was carried out by 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 stated in Section 3.2.

Shortcomings during initial development. One enhancement aimed at making *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.

Particularly relevant concepts for this enhancement were "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 virtually 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 should 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 identifier named `position` violates 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.

Decay. Subsequent extensions of the *CloneDetective* delivered an instructive example for the threat of code decay (Eick et al., 2001) 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 for 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 months after this change one could still find identifiers named `line` in almost all modules. Firstly, this did not 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 a concise identifiers named `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 Section 2). Students working on the program were not able to comprehend its original meaning in many places. They generally considered it a 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).

5.2. Naming in open source software

Suspicious vocabularies. Section 1 presented the results of an analysis revealing that approximately 70% of all characters of Eclipse's code base are identifiers. To gather further insights about identifier naming issues we further investigated the number of different identifier names. In the case of Eclipse 3.1.1 this yields the astonishing figure of 142,275 different names which clearly even outnumbers the number of words in the Oxford Advanced Learner's Dictionary. Naturally this high number stems mostly from usage of compound identifiers. The fact that identifiers in Java are commonly written in CamelCase (Sun Microsystems, 1999) allows the application of a simple heuristic to break the compounds in distinctive words. Counting only the atomic names still yields 11,482 different words. Considering the fact that speakers of English as second language need a vocabulary of around 5,000 words to understand academic texts (Tschirner, 2004) 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 unveiled that almost all identifiers are grouped in blocks with very similar names like: `frag`, `fragement`, `fragment`, `fragmentation`, `fragmented`, `fragmentname`, `fragments`, `frags`.

Top 10 composition elements. As part of our research on compound identifiers we analyzed the typical length of compound names in terms of number of atomic names used. While Eclipse has a limited number of identifiers consisting of more than 10 atomic elements the overall majority (>99%) consists of 1 to 10 elements. The distribution is shown in figure 4. Interestingly the groups of atomic identifiers (1 element) is only the fourth largest.

An important finding was made regarding the extremely high frequency of a limited set of atomic names: At least one name out of the ten most frequent atomic names (see fig. 5a) appears in almost *one third* of all identifiers.

One may argue that this is mainly due to generic words like *get* or *set* in the list of the most frequent words. Although this undoubtedly has an influence we found that such standard prefixes are less significant than expected. We manually adjusted the top-10-list by removing generic words (e.g. *get*, *set*) and moving up candidates that did not make it into the top 10 before (see fig. 5b). Counting the number of identifier names containing one of these words still lead to a number of 27,548 identifiers which was approximately 20% of all identifiers.

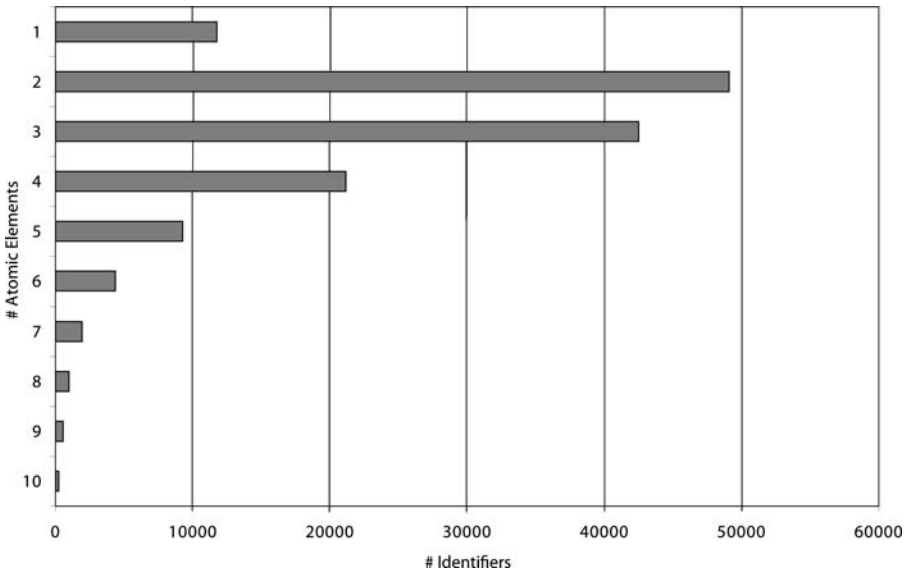

Fig. 4 Identifier length distribution

Fig. 5 10 most frequent atomic names: a) unadjusted b) adjusted

a)	b)
action	action
create	dialog
get	editor
is	file
name	java
new	label
page	name
set	page
to	text
type	type

As this experiment could be repeated with similar results on other software systems (see below) we conclude that the precise definition of the most frequent names has a strong impact on the overall naming and is therefore of paramount importance as well as a key opportunity to improve identifier naming and thereby program comprehension.

Validity of composition. Our rules concerning identifier composition are influenced by morphological research on natural languages. To make sure that these rules can be applied to programming languages we performed an analysis precisely determining the kind of compounds used in programming languages. To determine the word class of every atomic name we used the Java API for the Wordnet dictionary (Miller, 1995). Unfortunately the Wordnet dictionary does not contain all atomic names used in Eclipse. In addition to this, the word class is ambiguous for many words, e.g. the word *break* can be a verb or a noun. Therefore our results are limited to the names that could be identified unambiguously (25% of the atomic names and correspondingly 10% of all identifier names).

Table 3 Token analysis for Sun JDK 1.5.0

Type	#	%	*characters*	%
Keywords	638,266	11.9	3,102,312	14.5
Delimiters	2,432,591	45.4	2,432,591	11.3
Operators	368,838	6.9	463,342	2.2
Identifiers	1,675,492	31.2	13,596,623	63.4
Literals	245,586	4.6	1,850,645	8.6
Total	5,360,773	100.0	21,445,513	100.0

Fig. 6 Frequency of compound types in Eclipse

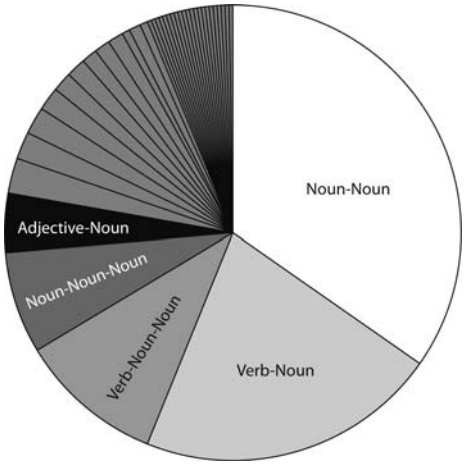

Figure 6 illustrates the most frequent kinds of compound identifiers and their number of occurrences. As expected, the most frequent kind are noun-noun compounds. Manual inspection of the verb-noun and verb-noun-noun compounds revealed that these are almost exclusively method names. Noun-noun-noun compounds and adjective-noun compounds are still somewhat frequent and covered by our composition rules, too.

Evolution of naming. To obtain a better understanding of the role of identifiers during system evolution we investigated the evolution of Eclipse's identifiers over a period of 38 months. Figure 7 shows that in terms of LOC Eclipse nearly doubled in size from version 2.0 (June 2002) to Version 3.1.1 (September 2005). Interestingly the number of different identifier names closely resembled this development; even topping it in the latest version. As expected the number of atomic names, the backbone of the vocabulary, did not grow with the same speed; in the same period it rose by only ca. 50%.

Further Sample Systems – JDK and Tomcat. A token analysis of Sun's 1.8 MLOC JDK 1.5.0 (see table 3) 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 65% of the code in terms of characters. The JDK features 64,652 different identifier names that are compounds of 9,659 different atomic names. 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 presumably synonyms.

Repeating the same analysis for Tomcat 5.5.14 (360 kLOC) confirmed the previous results. As table 4 indicates, identifiers accounted for about 30% of tokens or 62% in terms of

Table 4 Token analysis for Tomcat 5.5.14

Type	*#*	*%*	*characters*	*%*
Keywords	123,594	11.0	604,320	13.7
Delimiters	526,386	47.0	526,386	11.9
Operators	68,789	6.1	83,072	1.9
Identifiers	345,400	30.8	2,730,164	61.9
Literals	56,584	5.0	466,097	10.6
Total	1,120,753	100.0	4,410,039	100.0

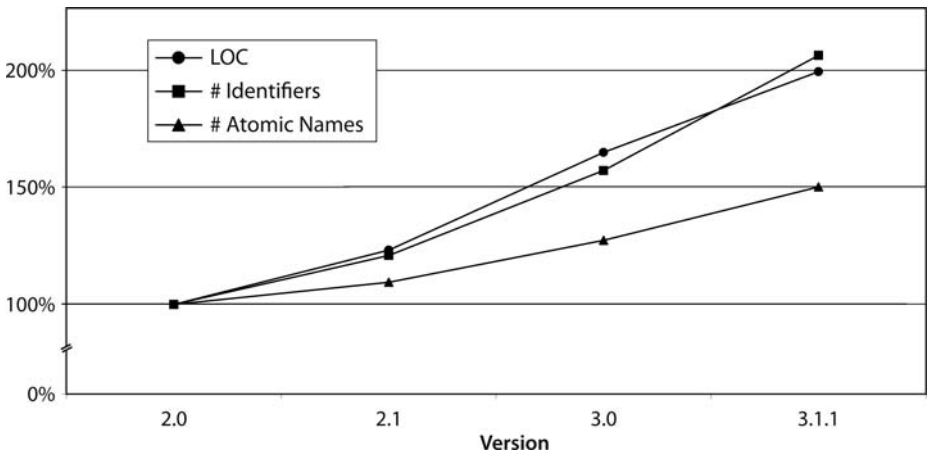

Fig. 7 Evolution of Eclipse (Version 2.0 ≡ 100%)

characters. The slightly lower percentage in terms of characters is due to the fact, that the average identifier name in Tomcat is slightly shorter than in the other examples. Tomcat's source code has 14,766 different identifier names composed from 3,302 different words. Our assumption about synonyms is again backed by the results of a manual inspection of identifiers.

Based on these results, it should be evident that identifier naming is a severe problem in real life software systems. Note, the shortcomings revealed can be both explained and avoided through the formal naming rules defined in 3.2. For example, the *fragment* synonyms found in Eclipse violate the formal consistency rule.

6. 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. But as a matter of fact, a carefully designed and implemented IDD will not reduce but increase the productivity of development, maintenance and quality assurance activities.

- *Development:* Developers can use the IDD to search for already existing identifiers instead of having to create or invent new ones. This reduces the risk of creating synonyms and helps to choose identifier names that follow existing naming patterns.
- *Maintenance:* The IDD speeds-up comprehension processes by enabling simple and fast lookups of the meaning (or at least a description) for identifiers. Vice versa 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.

6.1. 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 into 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 name-guided fashion. For example, developers should be able to look up a particular name in the source code, ask the tool for a list of all identifiers with this name, 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 are 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 representations of the dictionary.

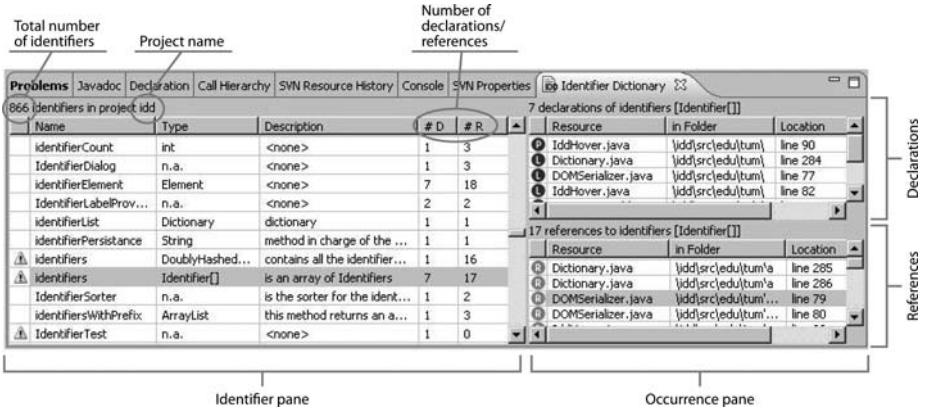

Fig. 8 Identifier Dictionary View

6.2. 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. Name 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. 8).

List views. All identifier names are listed with their 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 the corresponding name. On selecting a name, the right part of the identifier view displays two lists of occurrences of the identifier. The first list shows details on the various declarations of identifiers with the selected name and the second lists references to them. The icon in the left-most column indicates the type of declaration such as local variable, field or method. The other columns specify the exact location of the identifier. Double-clicking on an occurrence opens an editor window an sets the cursor to the specified location.

Warnings. During the collection process identifier names may become annotated with warnings icons (fig. 8) indicating potential consistency problems. Currently the IDD features two basic warnings: If two identifiers with identical name but different type are found, the

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

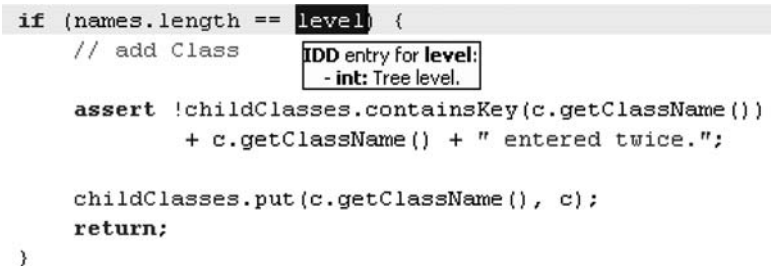

Fig. 9 IDD Hover Popup

identifiers are annotated to give a hint to a consistency violations. A typical example is the usage of the name `file` for objects of a class `File` and `String` type data objects holding the name of the file. The latter 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 are normally limited to local variables and private class members. IDD's detection of unreferenced identifiers works for all kinds of identifiers including methods regardless of their visibility.

Popups. Further assistance for developers is provided by *hover popups* that offer on-the-fly 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 (see fig. 9). By using this feature the developer can query the IDD without leaving the editor and switching to another view.

Auto-completion. 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 within the scope of the current editing location. Hence, it is usually not possible to auto-complete a variable declaration statement that starts with `int abso...` . The IDD extends Eclipse's auto-completion feature for all statements and expressions 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 may choose. The matching algorithm not only considers the name of the identifier but also its type (e.g. `int`).

Global name refactoring. 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 for arbitrary kinds 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. Because of this, the developer does not have to carry out extensive manual renamings to preserve consistency and conciseness of identifier naming while evolving the system. Clearly, manual renamings are not only tedious but they also easily lead to incorrect code.

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.

7. 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. With the help of a formal model we explained what proper naming actually means and gave various practical examples. The consistency, conciseness, and composition 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 programming tasks such as determining the name of an identifier within a declaration.

The identifier dictionary can obviously not 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. 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.

References

Pigoski, T.M. 1996 Practical Software Maintenance. Wiley Computer Publishing.

Lehman, M. 2003 Software evolution threat and challenge. Professorial and Jubilee Lecture (2003) 9th international Stevens Award, hosted by ICSM.

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

Sun Microsystems Code conventions for the Java programming language. Technical report, Santa Clara, CA (1999).

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

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

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

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

Sneed, H.M. 1996 Object-oriented cobol recycling. In: WCRE '96, IEEE Computer Society 169.

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

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

Rajlich, V., Wilde, N. 2002 The role of concepts in program comprehension. In: IWPC '02, IEEE CS 271.

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

Chen, K., Rajlich, V. 2000 Case study of feature location using dependence graph. In: IWPC '00, IEEE CS 241.

Shneiderman, B. Software psychology. Winthrop Publishers, Inc. (1980).

Anderson, J.R. 1995 Cognitive Psychology and its implications. W. H. Freeman and Co., New Jersey.
Wilson, J.Q., Kelling, G.L. 1982 Broken windows. The Atlantic Monthly **249**(3).
Hunt, A., Thomas, D. 1999 The pragmatic programmer: From journeyman to master.
Weinberg, G.M. 1971 The psychology of computer programming. Van Nostrand Reinhold Co.
Robertson, J., Robertson, S. 2004 Volere template v10.1. Requirements specification template, Atlantic Systems Guild.
Allen, F.W., Loomis, M.E.S., Mannino, M.V. 1982 The integrated dictionary/directory system. ACM Comput. Surv. **14**(2) 245–286.
O'Grady, W., de Guzman, V.P. 1996 Morphology: The analysis of word structure. In O'Grady, W., Dobrovolsky, M., Katamba, F., eds.: Contemporary Linguistics: An introduction. 3rd. edn. Longman, London and New York 132–180.
Costello, F. 2002 Investigating creative language: People's choice of words in the production of novel noun-noun compounds. In: Proceedings of the 24th Annual Conference of the Cognitive Science Society. 232–237.
van Rysselberghe, F., Demeyer, S. 2003 Evaluating clone detection techniques. In: ELISA 03. 25–36.
Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L. 1998 Clone detection using abstract syntax trees.
Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A. 2001 Does code decay? Assessing the evidence from change management data. IEEE Transactions on Software Engineering **27**(1) 1–12.
Tschirner, E. 2004 Breadth of vocabulary and advanced english study: An empirical investigation. Electronic Journal of Foreign Language Teaching **1**(1) 27–39.
Miller, G.A. 1995 Wordnet: a lexical database for english. Commun. ACM **38**(11) 39–41.

Florian Deissenboeck is a research assistant in the Software & Systems Engineering group of Prof. M. Broy at the Technische Universität München. Currently he works on his PhD thesis about software quality controlling. His academic interests lie in software maintenance, software product quality and program comprehension. He studied computer science at the Technische Universität München and the Asian Institute of Technology, Bangkok.

Markus Pizka received a Dr. degree from the Technische Universität München for his work on Distributed Operating Systems in 1999. During this work he extended GNU GCC, Linux and other Open Source Software packages. He lateron shared his experiences in compilers and reverse engineering with Microsoft Research, Cambridge and went on as a project leader for a large German software company. In 2001 he rejoined the Technische Universität München as an assistant professor and shaped a software maintenance curriculum. In 2003, he founded itestra, a German software consultancy providing commercial software reengineering services.