Consistent Consideration of Naming Consistency

Daniel Speicher, Jan Nonnen

Computer Science III, University of Bonn

{dsp,nonnen}@cs.uni-bonn.de

1 Introduction

Naming is essential for code quality and code comprehension. Essentially names are the glue that helps programmers to associate program elements with the concepts in mind. Program elements, representing real entities, should be named after those (e.g. `FlightSchedule`, `ParkingTicket`, `AnnualNetProfit`). For program structures, that arise for technical reasons, names are chosen according to corresponding structures in reality (e.g. `Observer`, `Factory`[1]). Therefore programmers are shaping code and names to fit their concepts.

Although the practical relevance of good naming is obvious, it is hard to create a theory of it. We can't compare the code directly with the concepts as we don't have direct access to the developers mind. As a substitute we analyse the lexical structure of the names in the code. Based on this we create hypotheses about the underlying conceptual structure.

2 Representation of Concepts in Code

Names of program elements are either atomic or compound lexical items. The meaning of compound lexical items is certainly some combination of atomic lexical items. So the meaning of the atomic lexical items is most important.

The arising question is, which of the occurrences of an atomic lexical item gives us the most information of its meaning, i.e. where the item is *introduced*. To motivate a heuristic to find these occurrences, consider the following examples: The class named `Coordinates` is probably representing the real world concept of coordinates, and the type `List` representing the concept of the abstract data type list. So the class `CoordinatesList` doesn't introduce any of the two parts of its name but depends on the the two other types. On the other hand if there is no type `Transformer` in our program, the class `CoordinatesTransformer` could be considered as introducing the concept of a transformer.

Motivated by these examples we define that, a type p named n *introduces* a non compound lexical item l: if $n = l$, or if l is contained in n and all other lexical items in n are already introduced. A program element p_1 is *naming dependent* on a program element p_2, if p_2 introduces a lexical item that is part of the name of p_1.

Our definition of introduces allows different types to introduce the same lexical item. These types are not necessarily presenting the same concept, forming a source of possible inconcistencies.

If an element e_2 has any dependency to an element e_1, one needs to understand e_1 before one can understand e_2. As a consequence if static and naming dependencies are in opposite directions, one needs to understand both elements at the same time. In this case we consider the dependencies to be inconsistent.

To summarize these consistency considerations, we demand that for each introduced lexical item there is exactly one program element, and that the naming depencies are compatible with the static dependencies. A naming dependency from p_1 to p_2 is *compatible with* the static dependencies, if there exists a path of static dependencies from p_1 to p_2.

3 Meta Model

The meta model that we implicitly used in the previous disucssions contains three spaces:

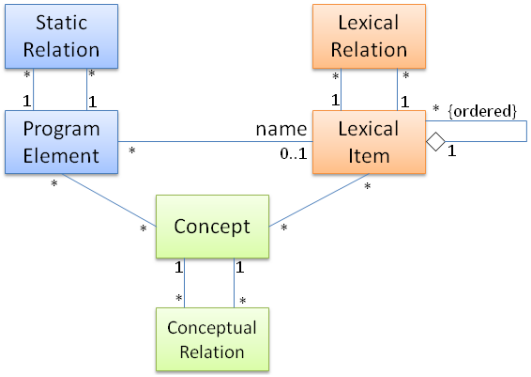

Figure 1: Meta Model

Program Space contains elements and relations describing static code dependencies. **Elements:** types, attributes, operations, behavior elements. **Re-**

[1] Here we mean *real* factories and *real* observers. Programmers are used to these as names of program structures, overlooking the metaphorical nature of them.

lations: calls, uses, extends, implements, depends on, control flow, data flow.

Lexical space contains program element names as well as parts of those. **Elements:** nouns, verbs, adverbs, adjectives, abbreviations. **Relations:** Rules for building compounds, fragments of sentences, sentences, abbreviations, derivations, and textual length.

Concept space contains our model of the programmers mental concepts. **Elements:** domain, domain entities, design pattern, roles in design pattern, architectural pattern, roles in architectural pattern, functional and non-functional requirements, metaphors. **Relations:** is a, is part of, is antonym of, is similar to, belongs to domain.

4 Consistency considerations

In section 2 we discussed demands for naming consistencies. In the following we try to formalize our approach.

Unambiguous mappings As we saw it is sometimes required that a mapping from one space to another is unambiguous, i.e. that there is exactly one image for each source.

Compatible relations Given relations R_1 and R_2 in the program, lexical, or concept space. With a mapping φ between two spaces, we call the relation R_1 *compatible (under φ) with the relation R_2 on the elements e_1 and e_2*, iff $e_1 R_1 e_2 \implies \varphi(e_1) R_2 \varphi(e_2)$.

Correlated characteristics Considering partial orders $\succeq$ or equivalence relations $\approx$, we can find mappings ψ into the real numbers, such that $e_1 \succeq e_2 <=> \psi(e_1) \geq \psi(e_2)$ or $e_1 \approx e_2 <=> \psi(e_1) = \psi(e_2)$ respectively. In these cases we can use correlation measures to asses the degree of compatibility between two relations.

5 Application

Our approach can be used to formulate (not to prove!) a variety of consistency demands.

Same level of abstraction Beck formulated in [1] the demand that behavior elements within one operation should be on one level of abstraction. With our words we could demand that the equivalence relation of being in one operation is compatible with the equivalence relation of having the same level of abstraction.

Conciseness corresponding to scope Martin formulated in [4] the idea, that the name of an operation should be shorter, if the operation is called from many other elements. In our words the order given by the number of incoming dependencies on the program space is compatible with the inverted order of the lexical item length.

Type postfixes Some lexical items define rules, resulting in patterns on program elements. E.g. an element's name ending with `List` indicates that the element should contain a abstract data type list of the elements of the type given by the prefix (e.g. `BoxList`). [5] contains studies on this correlation between type name and micro patterns [3].

Information $tf * idf$ is a established measure for the information that is given by the occurrence of a certain lexical item. If we apply this measure on the name of an operation on the one side and on the behavior of an operation on the other side, one could expect that the two orders are compatible with each other or at least correlated.

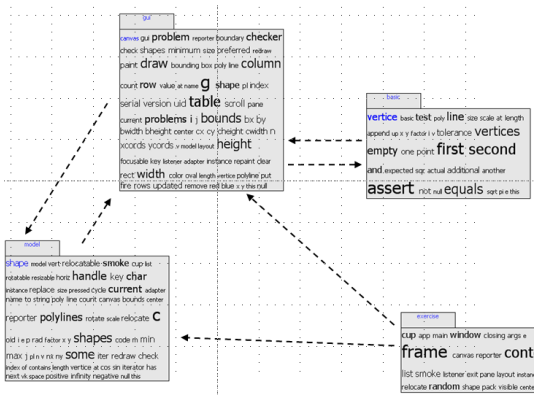

Figure 2: Our term dependency view. Indicating introduced items of a package in blue, and naming dependency between packages in an application.

6 Perspectives

We must elaborate how recent approaches like [2] to evaluate identifier quality relate to our approach.

The strict format is meant to encourage creativity, to systematically explore different flavours of consistency for their relevance. This could be achieved by studying the relevant elements and relations in the three spaces. After implementing these consistency checks, they can be validated and refined.

References

[1] K. Beck. *Implementation Patterns*. Addison-Wesley Longman, Amsterdam, 1 edition, Nov. 2007.

[2] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, 2006.

[3] J. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, volume 40, pages 97–116. ACM New York, NY, USA, 2005.

[4] R. C. Martin et al., editors. *Clean code: a handbook of agile software craftsmanship*. Prentice-Hall, 2009.

[5] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 67–76, 2008.