How Many Realities Fit Into a Program?

Notes on the Meaning of Meaning for Programs

Daniel Speicher, Jan Nonnen, Holger Mügge

University of Bonn, Computer Science III
{dsp,nonnen,muegge}@cs.uni-bonn.de

Abstract. Programs are written in programming languages with a certain well defined semantics that describes how an interpreter or a machine will operate based on the program. Higher level programming languages and especially object-oriented programming languages encourage programmers to write programs that contain knowledge and have meaning in an additional sense. This meaning of program elements, their identifier and the terms from which identifiers are built is the topic of this paper. Programs gather knowledge of different realities. There is at least an application domain and a technical domain. If we want to make the knowledge within a program more explicit and accessible, we need to differentiate, which program element refers to which domain.

Introduction

The first object-oriented programming language "Simula" presented its self-conception of its mode of denotation, i.e. how it relates to what it represents, in its name: Programs simulate reality. Objects as defined by the language, behave in a certain sense like real things, but they are no real things. If we consider programs simulations, programs are closed worlds simulating something that could be in a real world, but without any connection between these two worlds.

A simulation represents the world that it simulates. So the meaning of the program is this represented world. And the knowledge about this simulated world within this program is not hard to find: It is just the whole program. In the following we present some thoughts commenting on the question what it means for a program to mean something. We hope to contribute thereby to a deeper understanding, how knowledge can be found within programs. Our comments consider the question in three major steps:

- What is meant by the program? Is meaning in general best understood, if we consider it as representing some reality?
- How does the program mean something? Is the mode of denotation always straight representation as it is the case for simulations?
- How can the meaning of parts of the program be identified? What are current approaches to identify knowledge in programs?

What is meant by the program?

Approximation: Concepts are meant. Ratiu and Deissenböck report in "How Programs Represent Reality (and how they don't)" [9] about a successful approach to identify semantic defects in programs. Ideally a concept has a unique name which is used as identifier for the program element that represents the concept, e.g. the concept of an *account* (that the reader has in mind as soon as its name is mentioned here) is represented by a class `Account`. A defect in the sense of the paper is a violation of one of the one-to-one relationships between program elements (program space[1]), the lexical elements in their identifier (lexical space) and the corresponding concepts (concept space). Program elements represent concepts and that is their meaning. The title of the paper implicitly identifies the concept space with reality. As a representation of the concept space the authors used the WordNet ontology [3] and mapped portions of it onto the program code via graph matching, improved in [10].

Things are meant! Is it really concepts that are meant by the program? We consult the philosophical subsection of linguistics that discusses the meaning of meaning: Semiotics. Anything that may have a meaning is called a *sign* in semiotics and relates to three spaces (called levels) [7, p.24]:

> "In considering signs, their relationship to the cognitive world and to the world of things, it is important to distinguish clearly between three different level of observation: (a) the **linguistic** level of signs (words, sentences); (b) the **epistemological** level of cognitive correlates (concepts, propositions, etc.); and (c) the **ontological** level of things, truth values, and facts."

Obviously the linguistic level can be seen as our lexical space and the epistemological level as our concept space. As the program space can not be identified with the ontological layer we need to add a separate space: Reality. We will prefer reality as the target space of meaning but not strictly exclude the concept space. Soon we will see that the distinction between concept space and reality is especially important, if the program interacts with reality.

Given a class `SalesContract`, "sales contract" has a meaning and "sales" and "contract" may have a meaning as well. In addition the class itself is a model of a contract and most of its attributes and methods are models of the state and behavior of the real contract. Therefore we may as well consider classes, attributes and methods as signs. If we want to be brief in the following, we use *program signs* to refer to these signs in the program space as well as to the signs in the lexical space, i.e. to program elements and terms.

Some programs do not represent. There are signs (in the sense of linguistics) that have meaning without representing something. Imagine seeing an inattentive person coming closer to a crowded street so that you fear for her safety.

[1] [9, 10] used the term "layer", but for this paper "space" is more instructive.

Shouting "hey!" towards her has the clear meaning to get her attention and to make her aware of the situation, but the word does not represent something. Likewise some programs just contain a textual representation of yelling at the interpreter or virtual machine.

For object-oriented programs it is quite natural to consider them as representation of something, as they consist of objects. But in general a programmer might not have had the intention to *represent* something with a program. The primary goal of a program is to *influence* a machine to behave in a certain way. The text might be written with the question in mind "What do I need to write to make the machine do this or that?" And the answer can be meaningless but effective: Sending a notification not because the event occurred, but because we expect a certain reaction - Programs that rewrite themselves - Provoking buffer overflows - Letting Prolog predicates fail to provoke backtracking. We suggest that one definition of "hacking" (in the pejorative sense) could be, that the program tries to instruct the compiler without creating at least partially consistent meaning.

Instrumental notion of meaning. Is it possible to think about meaning without the notion of representation? Keller elaborates in [7] besides the *representational* notion (based on Aristotle and Frege) the *instrumental* notion (based on Plato and Wittgenstein) of meaning. From the instrumental perspective the core question is: What makes a sign a good instrument for communication? Or, to transfer a question stated by Plato (cf. [7, p.viii, p.47]) to our context: "When a programmer writes *this* program element or term and has *that* thing in mind, how is it possible that another programmer, who reads it, knows that the first programmer has it in mind." Elaborating a remark by Wittgenstein Keller explains that the *consistent use* of a sign allows for understanding: "[. . .] *the meaning of a word in a language, L, consists of the rule of its use in L.* [. . .] If you know how a word is used – if you know the rule of its use in the language L – you know everything there is to know." (cf. [7, p.51], emphasis in the original.)

The "rules" Keller refers to are regularities to be processed by the minds of the human users of the language, not by machines. Such rules for program signs can only have evolved for program signs that are used within a larger culture of developers. We suggest that e.g. the use of "open" and "close" is governed by rules of this strength. If an object provides *open* and *close* operations, we expect that it allows for a certain kind of interaction which is only valid after the *open* operation and before the *close* operation is executed. Developers understand the meaning of "open" and "close" without any notion of representation. Notice as well that the notion of "certain kind of interactions" is too imprecise to be checked by a machine but good enough for human minds. The rest of the rule is a well-formedness condition that is perfectly understandable to machines.

Structural Knowledge. We can make part of the meaning (in the instrumental sense) of program signs explicit by adding explicit definitions of the code structures in which they can be used to our code. Knowledge about structures is already very useful on its own and there is a whole body of work about mining and representing this knowledge. Examples are architectural patterns, design patterns and programming idioms. Design pattern use many terms in analogy to real things (e.g. the *Product* creating *Factory*, *Visitors*, *Observers*), but the rules of the usage of these terms in programs have become so strict, that we can say, that their meaning now lies in their usage.[2] From a pragmatic perspective we may – giving up philosophical precision – use the encoded structural regularities as a representation of the (instrumental) meaning of program signs and call the space of these regularities the *structural domain*.

How does the program mean something?

Objects in programs as well as things in reality have identity, state and behavior. To implement simulations one can follow a *natural modeling* approach: Represent the state of the real thing by attributes of the object and behavior by methods. In this case the implementation can be an accurate model of the represented reality. If the program does interact with reality the natural modeling approach fails and programs are for good reasons systematically distorted representations.

Non-identical Individuals and Specifications. In a perfect model there would be exactly one object for one thing. Yet, already a standard class like `File` from the package `java.io` expresses that it is not identical with the thing it represents. It is possible to interact with a `File` object even when the method `exists()` returns `false` or after calling `delete()`. While `delete()` represents a meaningful operation on the real thing, `exists()` is only meaningful on the object. A simulation would not need to care for the lifespan of a real file and the existence of a file could easily be represented by the existence of an object.

For many programs individual things are not important, but sets of individuals with common properties. We can call the object that specifies such a set following [4, p.81] a *specification*. Such an object has attributes that correspond to state of all things in the set, but adds methods that provide information about the set. For example a configuration for the rocket family Ariane could be represented by an `Ariane` object that one could ask for the number of individuals and a call to `new Ariane(5, “ECA”).numberRocketsBuilt()` would yield 33.

Peter Coad: Archetypes. Peter Coad's Archetypes introduced in [1] are mainly a precise distinction of entities with respect to their mode of denotation. Objects of a class of the archetype «party», «place» or «thing», represent a single party/person, a single location or a single thing. Objects of a class with the archetype «description» represent more than one thing, i.e. it is a specification in the sense we just introduced. Objects of an «moment-interval» class represent an event or process that takes place at a certain point in time or extends over

[2] Keller elaborates that the representational notion of meaning does not allow to explain the evolution of meaning, while the instrumental notion does.

a certain time. Objects of a «role» class finally collect any information about a «party», «place», «thing» or «description», which are only relevant in the context of one or more «moment-interval»'s. These distinctions are very relevant for the process of modeling, but these archetypes can only be understood, by taking reality into account - a dimension that is not accessible to the compiler.

Mirror, Window, Itself. There are two basic modes an object can interact with the thing it represents. They result in two different modes of denotation. First, the object may interact via a boundary with the real thing. This allows the remaining objects to call methods on the object as if they were operations on the real thing, i.e. the object is a *window* to the real thing. Second, other objects indirectly interact with the real thing and represent the changes in the object, i.e. the object is a *mirror* of the real thing. Programs that contain mirrors of windows typically contain as well objects that do not represent any real thing. Such an object can be just *itself.*

In a simulation the ability of a rocket to be launched can be represented by a method `Rocket.launch()`. In a program *controlling* a real rocket the same signature can be used, if we consider the object to be a window to the thing. Unfortunately there can be many obstacles during a rocket launch - even in the communication between the program and the real rocket so that Spolsky's law [11] applies: "All non-trivial abstractions, to some degree, are leaky." The abstract representation of the launching process by the `launch()` method is in many cases a good model but as obstacles may occur, the system needs to handle them. Therefore it makes sense to reduce the illusion of a direct access to the real thing, e.g. by changing the names of class and method to `RocketProxy.initiateLaunch()` or by declaring exceptions. Then the class name expresses that the object is not the rocket itself but a proxy for it. The method name makes clear that the method invocation does not execute the launch, but only initiates it, which might fail. The goal of a reconstruction of the model would be to find out that "rocket" is a thing that can "launch". In addition the reference to the *Proxy* design pattern should be detected, as well as if "initiate" has an (instrumental) meaning.

In case the class `Rocket` is a mirror of a rocket, it probably has no method `launch()`. This method will be in another object that uses an object of class `Rocket` to store for example the start time of the rocket by invoking a method `Rocket.setLaunchInitiated(Time)`. The distortions in mirror objects are larger, but the same amount of knowledge should be represented. Behavior of the real thing is often represented as state. Sometimes behavior that belongs to the real things is not present in the mirror object but in the objects collaborating with it.

Systems automating the real world. Isoda describes in [6] that for simulations models can be build by natural modeling. Unfortunately this already not true anymore for systems that *automate* the real world, e.g. that facilitate business processes. If the software would contain a natural model of the automated real world, it would need to contain itself. In addition the natural modeling approach would lead to classes unrelated to the business process under consideration, or class that are related but unnecessary for the implementation, or to methods in necessary classes that are confusing (cf. [6, p.159]). Therefore already [4, p.31] recommended: "For real-world objects, create system representations of those objects that respond to events rather than initiate them. They should be maintaining a representation of the object, rather than initiating activities." In our terminology: These objects should be mirrors. If developers want to make this explicit, the can name a class instead of `Customer` for example `CustomerData`. Note that "customer" refers to the application domain and "data" the technical domain of the (virtual) machine.

Boundary, Control, Entity. The distinction of classes using the UML stereotypes «boundary», «control», «entity» corresponds to differences in the mode of denotation. Entities are typically mirrors of real things. Boundaries as elements of the user interface are typically windows (in our sense) to virtual elements on the screen. User and developers can easily ignore the virtuality of these elements ("mediated immediacy") and typically do so. Still there is more about this object than internal state. Classes of these stereotypes and their names tend to refer to different realities: Entities refer to the application domain. Boundaries refer to the virtual reality of user interfaces often mimicking different real devices. Controls might represent processes in the application domain, but add further more technical processes.

How can the meaning of parts of the program be identified?

Debugging Method Names. Høst and Østvold presented in [5] an approach to mine common properties of methods following certain naming schemes. They used properties like *returns-void*, *creates-own-type-object*, *catches-exception*. The naming schemes had not been predefined, but they explored for which naming schemes strong statistical evidence was found. This knowledge was precise and reliable enough to identify methods with inappropriate names. This work can be seen as an example how knowledge about the structure of methods can be found in names.

Mining identifier "ontologies". Falleri et al. suggested in [2] to create a WordNet like structure from method names. The method names are split into terms and tagged with their part-of-speech. In a second step the terms are sorted by linguistic dominance. Intuitively a word a dominates a word b, if b renders a more precisely. For example "getNextWarning" is parsed into ("get", "next", "warning"). Adding the part-of-speech yields the term list (("get", Verb), ("next", Adjective), ("warning", Noun)). Finally dominance sorting gives (("get", Verb), ("warning", Noun), ("next", Adjective)). If a term list is a prefix of another term list, it is considered to be a generalization of the later. Longest common prefixes of these term lists are then regarded as entities in their own right. For example if we have "getPreviousWarning" as well, "getWarning" is added and considered a generalization of the two other methods.

Location of Origin for Terms in Names. To identify to which reality a term in a name refers, it is useful to find out where in the program it originated. Considering the identifier “WindowFactory” we will probably find both terms to be originating from the same domain. Either both were first used in GUI code or both were first used as part of the model of the application domain. In the identifier “FactoryWindow”, the term “factory” might stem from the domain model, while “window” is an GUI term. The “factory” would represent a real factory and the “window” would be a virtual entity in our program.

In [8] we reported about an extensive exploratory study that defined and evaluated a heuristic to identify *introduction locations*. To subsume both notions of meaning we defined a class c to be an introduction location for a term t, "if the meaning of t can be understood by reading the code in c." Our heuristic already had a precision of 75%. Based on the theoretical reflections presented in this paper, we can make our definition even more precise hopefully leading to further improvements of our heuristic: Program elements that are models represent. Such an program element should be seen as the introduction location for all terms in its name as long as there is no better candidate, i.e. with fewer terms in the name. For terms used in names of program elements that do not represent but do have meaning in the instrumental sense we have to select the location that illustrates the rules of its use best. In case we added explicit definitions of the structures in our code to our code base, these definitions should be regarded as the introduction locations.

Conclusion, Research Objectives, Approaches

We discussed the *representational* and the *instrumental* notion of "meaning" for program signs. Typical programs contain program signs representing portions of at least one *application domain* and some *technical domains*. Other program signs allow for understanding by following regularities in their use. As far as these regularities are structural, we can define them as elements of the *structural domain* and consider the signs as representing them. This leads us to the research objectives to identify these regularities and to identify which program signs represent the same domain.

The question whether domains are systems of concepts in mind or of things in reality is not purely philosophical. Some program elements that *represent* a real thing can as well indirectly *interact* with this thing - a semiotic relation that is specific for computer programs. An object interacting with a real thing that it represents is typically either like a *window to* or a *mirror of* it. As windows and mirrors do not perfectly represent the thing, it is a research objective to reconstruct better models from them. Another objective is to automatically identify objects that do not represent individual things but sets of similar individuals.

Once we correctly identified the meaning of program signs representing the same domain, the question of the *meaning of the whole* arises. We expect the composition to have meaning in the instrumental sense. As such it is rather independent from external realities, and may dynamically evolve over time. The same is true for program signs representing structures. This leads to the research question, how such an *evolution of meaning* can be identified.

As a basis for further research, we suggest to conduct thorough *code based case studies*: Can an expert developer identify the represented domains for all program signs in sample objects? Can she identify the modes of denotation for objects and correct the imperfections? As result we will get annotated code bases, that can be used as benchmark for algorithms and as prototypes of supplemental knowledge representation.

We demonstrated that certain objects have to be for good reasons imperfect models. Therefore we suggest to *revisit the modeling literature* from the last century with respect to their recommendations, how to translate reality into code. Where modeling defects are created systematically, we expect that some meaning can be restored based on manually defined or even mined rules.

The algorithms mentioned in the previous section are promising candidates for answering the research questions stated: Terms that are introduced (in the sense of [8]) in the same package probably refer to the same reality. Complementary [2] identifies regularly reoccurring combinations of terms that we can expect to have structural meaning. Pattern mining like the approach in [5] on the level of methods even identify some of the structural regularities. In case we have an ontology as a second representation of the represented reality the approach presented in [9, 10] may be used to identify the program signs that refer to this reality and to find modeling defects. Applying these technologies on software repositories gives a first approach to the identification of the evolution of meaning.

References

1. P. Coad, E. Lefebvre, and J.D. Luca. *Java modeling in color with UML: enterprise components and process*. Java Series. Prentice Hall PTR, 1999.
2. J.R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic Extraction of a WordNet-Like Identifier Network from Software. In *ICPC*, pages 4–13, 2010.
3. C. Fellbaum. *WordNet: an eletronic lexical databse*. Cambridge MIT Press, 1999.
4. S. Gossain. *Object modeling and design strategies: tips and techniques*. Advances in object technology. Cambridge University Press, 1998.
5. E.W. Høst and B.M. Østvold. Debugging Method Names. In *ECOOP*, 2009.
6. S. Isoda. Object-oriented real-world modeling revisited. *Journal of Systems and Software*, 59(2):153–162, 2001.
7. R. Keller. *A theory of linguistic signs*. Oxford University Press, 1998.
8. J. Nonnen, D. Speicher, and P. Imhoff. Locating the meaning of terms in source code, research on "term introduction". In *WCRE*, 2011.
9. D. Ratiu and F. Deissenböck. How programs represent reality (and how they don't). In *WCRE*, pages 83–92, 2006.
10. D. Ratiu and F. Deissenböck. From reality to programs and (not quite) back again. In *ICPC*, pages 91–102, 2007.
11. J. Spolsky. The law of leaky abstractions, 2002. [Online; accessed 01-August-2011], http://www.joelonsoftware.com/articles/LeakyAbstractions.html.