Towards an Explicit Intentional Semantics for Evolving Software (accepted submission to ASE'98 doctoral symposium)

Author: Kim Mens
Thesis advisor: Theo D'Hondt
Vrije Universiteit Brussel
Programming Technology Lab (PROG)
Pleinlaan 2, B-1050 Brussel, Belgium
kimmens@vub.ac.be

Abstract

The subject of my PhD work is the study of software engineers' intentions *and the importance of using the information provided by such intentions during the software engineering (SE) process. More specifically, we will study how explicit software intentions can contribute to a better understanding of the software, and how automated reasoning about explicit software intentions can facilitate many software engineering activities, and software evolution in particular.*

1. Introduction

It is generally acknowledged that a lot of software[1] today is difficult to understand, maintain or adapt, hard to reuse, difficult to evolve, and so on [1, 3, 6, 7, 13]. This is partly due to the fact that most software contains a lot of hidden assumptions. The software reveals only *how* things will work, and (implicitly) *what* will happen, but provides little or no information on the *intentions* of the engineers that built the software (e.g. *why* something was constructed in a certain way). Even when the software does contain such information it is most often implicit or described informally in the software documentation [18].

Our contribution will be to make a first step towards a kind of intentional 'semantics' for software in which this kind of information can be expressed explicitly, preferably in a computable and declarative way, and to show how automated SE tools can use this information to make software more 'manageable'. We do *not* intend to develop a complete formal semantic model, but rather to study the use of intentions in automated SE tools.

To restrict the scope a bit, we focus on the domain of evolution of object-oriented (OO) software[2], and set out to prove the following thesis:

> **Thesis:** *Automated reasoning about explicit information on the intentions of software engineers allows to build more powerful tools for software evolution. (More powerful in the sense that they can draw stronger conclusions by reasoning not only about the software but also about higher level conceptual information, i.e. the software intentions.)*

We admit that this thesis is still somewhat too broad and needs to be made more precise. For example, the kind of software evolution *tools* we are particularly interested in are tools for detecting evolution conflicts. We will try to show that conflict detection tools using intentional information can be made more powerful in the sense that they can detect more conflicts. Also, we need to make more precise *how* intentional information will *allow to* do this.

2. Intentions

When constructing a software artifact, a software engineer constantly makes important and less important choices and decisions. These decisions are typically based on and motivated by various assumptions about the problem domain, about the software requirements[3] (functional as

[1] We explicitly use the term 'software' throughout this paper instead of the word 'code' or 'program', because we believe the same research problems and solutions are also relevant to artifacts in other phases of the software life cycle such as requirements, architecture, analysis and design.

[2] We choose evolution and OO because of our background in these domains [17] and because they pose some non-trivial and important problems.

[3] Note that not all software requirements are known beforehand. For example, it often happens that a software engineer discovers new important requirements during the software engineering process.

well as non-functional), about other software artifacts with which the artifact under construction should co-operate or upon which it should build, and so on...

All these assumptions and the associated intentions of a software engineer when making decisions, usually are not captured explicitly in the software. Only the results of the decisions that were made can be found in the software. In the best case an engineer writes down his or her intentions on paper or in the software documentation in natural language, or uses certain conventions, software patterns or follows certain style guidelines from which some intentions can be derived implicitly. (For example, using a strategy design pattern might express a designer's intention to make an important algorithm easily replaceable by a variant [5], or "best programming patterns" might be used to communicate programming intentions [2].) Most intentions however, e.g. *why* software was constructed in a certain way, are difficult or impossible to extract from the software. (As opposed to information on *what* the software does, and *how* it works, which usually can be derived implicitly or explicitly from the software.) Therefore, we think there is a need for making these intentions explicit.

Intuitively, we could define a software intention as any kind of information on the purpose of the software, that is not explicitly contained in the software itself. In other words, an intention is a meta description of the software that motivates why the software is constructed in a certain way. But not any meta description is a software intention: only those meta descriptions that link software artifacts to the 'hidden assumptions' are software intentions.

> **Definition:** *A software intention is a meta description of the software that links software artifacts to the 'hidden assumptions' made by a software engineer (about the problem domain, about the software requirements, about the purpose of related and co-operating software artifacts,...).*

One of the reasons why software engineers are unable to adequately document their intentions is that SE tools and notations provide insufficient support for expressing intentions in a more explicit, formal and disciplined manner. We feel that such information should be incorporated in automated SE tools and that it can play an important role to facilitate SE activities in general, and software evolution in particular. However, although we want to express intentions in a formal way, we want a notation that is simple enough to be used and accepted in practice, and easy to be manipulated in tools. We claim that a need exists for building SE tools that can reason automatically about such explicit intentions.

Our claim is supported, amongst others, by [11], where it is argued that software evolution currently suffers from a lack of intentional information: when the original software engineers' intentions are insufficiently documented, their continued involvement is needed to enable later engineers to learn their way through the software system and to better understand the assumptions behind the system's design. As this may be too time-consuming or simply impossible when the original software engineers are not available anymore, a more accurate documentation of the engineers' intentions is required. Lehman[4] also agrees that software engineers' hidden assumptions should be made explicit in the software, preferably in a structured and machine-processable form, to facilitate change management during software evolution [9]. He argues that at all stages of the software life cycle,"attempts must be made to recognize, capture and record assumptions, whether explicit or implicit, in design and implementation decisions, as must any dependencies and relationships between them".

Therefore, we assume the following research hypothesis.

> **Research hypothesis:** *Many SE activities (such as software maintenance, adaptation, evolution, reuse, re-engineering, reverse engineering,...) benefit by intentional information of the software engineer.*

We motivate this research hypothesis, by arguing that some of the technical problems that hinder these activities could be solved more easily if one would have more intentional information of the software engineer. Some of the technical problems are:

1. understanding[5] the purpose of software artifacts, as well as why they were constructed in a certain way;
2. understanding the dependencies and relationships between different software artifacts;
3. detecting and solving conflicts when changing, adapting, evolving or reusing software artifacts;
4. traceability of software artifacts.

It is clear that the first two problems immediately benefit by more intentional information. Solving the second problem is important to be able to assess the impact of making changes to certain software artifacts on the other software artifacts. The third problem is a special case of the more general problem of *compliance checking*: checking whether some evolved software artifact conforms to what is expected from it, i.e. does it work together correctly with other software artifacts, are the assumptions that it makes

[4] Lehman studies the *laws of software evolution* and their implications to improve software processes dealing with evolution.

[5] Although we think that software intentions can clearly contribute to the research domain of software comprehension, our focus will be more on the use of intentions to enhance software evolution tools.

and that are made about it valid, does the software artifact respect the original intentions, ... ? It should be at least intuitively clear that compliance checking can benefit by more intentional information. Finally, *traceability* problem should become easier when one has more intentional information. According to [15], traceability in analysis comes down to"justifying the existence of a given result by tying it back to the stated goals and objectives" . This information could be expressed by explicit intentions.

We will try to validate this research hypothesis in practice by showing that automated reasoning about software intentions does not only make it possible and easy to build automated SE tools (and tools for checking evolution conflicts in particular), but also allows us to draw stronger conclusions than without that information. This immediately proves our thesis as well.

3. Approach

In this section we summarize the approach we will follow to obtain the results described above. Our approach could be described as a "bottom-up approach with a top-down vision". Our ultimate goal is to show that automated SE tools can use explicit intentional information to make software more manageable. However, to simplify things at first, we limit the scope by looking at the problem of evolution of OO software. Later we broaden the scope again and show that the results are also valid for other SE activities (than software evolution) and other programming paradigms (than OO).

Instead of immediately trying to build a general formal model of software intentions, we focus on a particular kind of intentions first and study what extra power they can provide. Although we still have to complete our literature study and make a categorization of the kinds of intentions that are most promising, we think it would be interesting to look at those intentions that can be expressed in terms of *classifications* and relationships between these classifications.

3.1. Classifications

The idea of a *classification* is to group a collection of software artifacts together because they ought to be considered as a whole (from an intentional point of view). All artifacts in a classification typically share some important feature. For example, in a financial application it could be interesting to group all software artifacts dealing with "handling deposits" together in a single classification. This classification expresses the intention that all these software artifacts cooperate in achieving the functionality of handling deposits.

A software artifact can belong to different classifications and a single classification can contain many different kinds of software artifacts. A classification does not necessarily correspond to the classifications that can typically be found in the software. The only requirement is that the software artifacts in a classification share some functional (e.g. handling deposits) or non-functional (e.g. aspects such as "persistency" or "distribution") feature. As such, classifications express part of a software engineer's intentions, because they provide conceptual classifications of software items that may not be found in the software itself. Dependencies and relationships between classifications ("part of", "is a", causal relationships as well as negative relationships stating independencies) can provide even more important intentional information.

Intentional information on which software artifacts are grouped according to which classifications and what the dependencies between the different classifications are could be used in tools for dealing with software evolution conflicts. For example, if there is a conceptual dependency between two classifications, one could expect that this dependency is reflected in some way by the artifacts that are contained in those classifications. If this dependency structure is accidentally invalidated upon evolution, there is an evolution conflict.

3.2. Validation

After having chosen a particular kind of intentions to investigate in more detail, we perform some experiments to validate whether the proposed approach actually works (i.e. that intentional information based on classifications and dependencies between them can really be used to solve new and interesting evolution conflicts). We will build a prototype of an automated SE tool (more specifically, a tool for detecting evolution conflicts) and apply it to an industrial case study. We will try to merge our theory and tool with the existing reuse contracts methodology [17, 12], which is a proven methodology for dealing with evolution conflicts in OO software. We plan the following validation experiment:

1. identify and declare some classifications as explicit intentional tags about the case;
2. identify and declare dependencies between classifications as intentional information about the case;
3. implement and test conflict detection and compliance checking rules based on this information;
4. analyze how this approach extends the reuse contracts model (i.e. how it makes it more powerful).

Whereas the purpose of this experiment is to show that software evolution tools benefit by more intentional information, we also need to investigate what happens when the intentions themselves evolve.

3.3. Generalization

To generalize the obtained results we will study which other kinds of intentions can be expressed and how they can be used to build more powerful software evolution tools. Next we broaden the scope and try to show that the results are also valid for other SE activities (than software evolution) and other programming paradigms (than OO). To conclude the thesis we hope to be able to show the generality of our research results by showing that existing "hard" semantic techniques which also declare a kind of intentional semantics, can be expressed with our approach as well.

4. Related Work

Let us conclude with enumerating some related work that seems relevant to the subject of the proposed thesis.

4.1. Program Comprehension Research

Program comprehension research results might provide interesting clues as to which kinds of intentions are useful to enhance the comprehensibility and evolvability of software. Although current program comprehension research fails to provide a clear picture of comprehension processes with respect to specialized tasks such as software evolution, some existing research results do indicate which kind of information is considered important by engineers when trying to understand software constructed by other engineers [19]:

- *Software-specific knowledge* relating to functionality, software architecture, the way algorithms and objects are implemented, and so on.
- Information on the *what*, *how* and *why* of software artifacts. [10] identifies three kinds of hypotheses people make when trying to comprehend a piece of software:
  - 'why conjectures' hypothesize the purpose of a function or design choice
  - 'how conjectures' hypothesize the method for accomplishing a (program) goal
  - 'what conjectures' hypothesize what the software does.
- Used *styles* and *conventions* ('rules of discourse') such as coding standards, algorithm implementations, expected use of data structures, and so on. Experiments [16] have shown that unconventional algorithms and programming styles are much harder to understand, even for experts.
- Information on the *control flow* and other dependencies (e.g. data flow) in the software. For example, [14] found that when code is completely new to programmers, the first mental representation they build is a control-flow program abstraction.

4.2. Intentional Programming

Microsoft researchers are investigating the concept of 'intentional programming', which seems closely related to our work, as it is also based on making intentions explicit in the software. They agree with us that "much of what makes programming[6] costly and time-consuming, including the declaration of design intentions, the identification of invariants, the alternatives which were not chosen, the overall structure, the dependencies ... and so on are either not encoded at all, or not encoded in a machine understandable form" [18]. However, whereas we see intentions as a kind of meta description on top of the software, they introduce intentions as a new programming abstraction which can actually be executed.

4.3. Features

We informally defined *classifications* as collections of software artifacts that need to be considered as a whole, because they share an important 'feature'. So classifications can be identified by identifying the important features. [7] provides some examples of (functional) features and defines a 'feature' as "any distinguishing characteristic of a software system that customers or reusers can use to select between available options". The FODA methodology [8] considers distinct types of features: operational, non-functional, development,... [4] defines a feature as "the difference that makes the difference" and provides some guidelines for identifying features.

4.4. Other Related Work

In the research areas of program understanding, design theory and knowledge based SE, many systems have been described that represent programming knowledge in one way or another. We need to investigate how these kinds of knowledge relate to software intentions.

5. Acknowledgements

Thanks to Kris De Volder, Steve Easterbrook, Carine Lucas, Tom Mens, Tom Tourwé, Bart Wouters and Roel Wuyts

[6] Note that we did not focus on the programming level only, but also on the other phases of the software life cycle.

for their comments on this paper and providing helpful comments on it. I also thank these proof readers, my thesis advisor Theo D'Hondt and Patrick Steyaert for guiding me in the quest for an interesting research topic.

References

[1] M. Aksit, B. Tekinerdogan, L. Bergmans, K. Mens, P. Steyaert, C. Lucas, and K. Lieberherr. Adaptability in object-oriented software development. In *Special Issues in Object-Oriented Programming, Workshop Reader of ECOOP'96*, pages 5–52. dpunkt.verlag, 1997.

[2] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[3] L. Bergmans and P. Cointe. Composability issues in object orientation. In *Special Issues in Object-Oriented Programming, Workshop Reader of ECOOP'96*, pages 53–124. dpunkt.verlag, 1997.

[4] R. Creps, C. Klinger, M. Simos, L. Lavine, and D. Allemang. Organization domain modeling (odm) guidebook version 2.0, 1996. Informal technical report for Software Technology for Adaptable, Reliable Systems. STARS-VC-A025/001/00.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.

[7] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addisson-Wesley, 1997.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[9] M. Lehman. Software's future: Managing evolution. *IEEE Software*, January/February:40–44, 1998.

[10] S. Letovsky. Cognitive processes in program comprehension. In *Proceedings of the First Workshop on Empirical Studies of Programmers*, pages 58–79. Ablex Publishing, Norwood, N.J., 1986.

[11] K. Lieberherr. Workshop on adaptable and adaptive software. In S. C. Bilow and P. S. Bilow, editors, *Addendum to the OOPSLA'95 proceedings*, pages 149–154. ACM Press, 1995.

[12] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1997.

[13] C. Pancake. Object Roundtable, The Promise and the Cost of Object Technology: A Five-Year Forecast. *Communications of the ACM*, 38(10):32–49, October 1995.

[14] N. Pennington. Comprehension strategies in programming. In *Proceedings of the Second Workshop on Empirical Studies of Programmers*, pages 100–112. Ablex Publishing, Norwood, N.J., 1987.

[15] K. Rubin and A. Goldberg. Object behaviour analysis. *Communications of the ACM*, 35 (9):48–62, September 1992. Special Issue on Object-Oriented Methodologies.

[16] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[17] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285. ACM Press, 1996.

[18] C. Symonyi. Intentional programming — innovation in the legacy age, 1996. Notes of presentation at IFIP WG 2.1 meeting.

[19] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.