Annotating Evolving Software with Explicit Intentions

Kim Mens

January 26, 1998

Abstract

Our position statement[1] is that current software evolution techniques suffer from a lack of documentation on software developers' intentions and that mechanisms to support evolution can vastly be improved by making these intentions explicit in the software. We provide an intuitive definition of "intentions", explain which kinds of intentions can be distinguished, discuss how such intentions can be made explicit in the software by attaching annotations to software artefacts and argue how such information can provide support during the software development process (and during software evolution in particular). Because of the preliminary status of this work, we mainly try to discover the important research questions to be answered and research topics to be investigated.

1 Introduction

Developing software that works is one thing, but developing software that can easily *evolve*, is another. When writing "evolutionary" software the "purpose" of the software should be clear so that it is easy to understand and change the software and so that the implications of making changes can be assessed better. Our claim that software evolution can be improved by making software developers' *intentions* more explicit is supported, amongst others, by [Lie95], where it is argued that software evolution currently suffers from a lack of information on such intentions. When the intentions of the original software developers are insufficiently documented, their continued involvement is needed to enable later developers 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 developers are not available anymore, a more accurate documentation of the developers' intentions is required.

Programme comprehension research results might provide interesting clues as to which kinds of intentions are useful to enhance the understandability and evolvability of software. Although current programme comprehension research fails to provide a clear picture of comprehension processes with respect to specialised tasks such as evolution, some existing research results do indicate which kind of information is considered important by developers when trying to understand software written by other developers [vMV95]:

1. software-specific knowledge relating to functionality, software architecture, the way algorithms and objects are implemented, and so on.

2. information on the *what*, *how* and *why* of software artefacts. [Let86] identifies three kinds of hypotheses developers 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 (programme) goal;
   - "what conjectures" hypothesize what the software does.

3. used styles and conventions ("rules of discourse") such as coding standards, algorithm implementations, expected use of data structures, and so on. Experiments [SE84] have shown that unconventional algorithms and programming styles are much harder to understand, even for experts.

[1] This position paper was submitted to the international workshop on the principles of software evolution, Kyoto, Japan, April 20 and 21, 1998.

4. information on the control-flow and other dependencies (e.g. data flow) in the software. For example, [Pen87] found that when code is completely new to programmers, the first mental representation they build is a control-flow program abstraction.

Some of the above types of information are more "descriptive" in nature and others have a more "intentional" character. For example, information on control-flow and software dependencies (e.g. the use of specialisation interfaces in [Lam93, Luc97]) merely describes what the software looks like and how it works, whereas, for example, most programming and design styles and conventions are intentional as they motivate *why* things are implemented in a certain way.

Whereas many current approaches try to enhance software evolution by reasoning about and making explicit *descriptive* information, the main contribution of this paper is our conjecture that software evolution can be improved much more by explicitly documenting and manipulating *intentional* information. Because of the preliminary status of this work, we will focus on identifying the important research topics to be investigated.

The next section provides an intuitive definition of "intentions" and explains the difference with conventional software descriptions. In the subsequent section we propose to make such intentions explicit in the software by attaching *annotations* to software artefacts. We conclude with a summary of the main research questions to be answered.

2 Intentions

When developing a piece of software, a developer constantly makes decisions regarding what the software should do, how it should work and how to organise it. Many authors [Lie96, SLMD96, KL92] agree that by explicitly documenting some of these decisions better, it becomes easier to understand the software and to reason about it. In analogy to Letovsky's distinction between *what*, *how* and *why* conjectures (section 1), we might make a similar distinction between different kinds of documentation on developers' decisions: documentation on *what* the software is and does and *how* it does things is clearly of a more descriptive nature than documentation on *why* things were done in a certain way. In the remainder of this paper, we will call the former kind of documentation *descriptions* and the latter *intentions*.

Many kinds of "descriptions" can be distinguished. A first distinction already mentioned above is the distinction between "what" and "how" descriptions. Pre- and post-conditions [Mey88] might be considered an example of *what descriptions*, as they document what a method or function does. [Lam93] uses specialisation interfaces to describe how the software works by documenting the important calling dependencies between groups of methods in a class. Another (more or less orthogonal) distinction that can be made is the distinction between "structural" and "behavioural" descriptions. Propagation patterns [Lie96] are a typical example of *behavioural descriptions* as they describe the behaviour of an (adaptive) programme as independent as possible from the actual class structure.

In literature, many examples of software descriptions can be found. Furthermore, lots of research efforts are currently being directed towards investigating how such software descriptions can promote software evolution. For example, [SLMD96, Luc97] describe object-oriented software with reuse contracts and show how many change propagation conflicts that arise upon evolution can be handled. And based on his notion of propagation patterns [Lie96] explains how to write *adaptive* programmes that are much more adaptable than conventional object-oriented programmes.

Whereas software descriptions document important decisions made by software developers and thus enhance understandability and adaptability of software, we feel that some essential information is still missing. Descriptions do not motivate *why* developers decide to do things in a certain way. We believe that by explicitly documenting and reasoning about the "intentions", software evolution techniques and mechanisms can be enhanced much more than can be achieved with mere descriptive documentation of the developer's decisions. [CNFG96, CNGM96] state that *deviations* from the intended purpose of the software should be documented clearly, because they are likely to give rise to evolution conflicts. Without explicit intentional information it is very difficult, or even impossible, to do this.

As opposed to software descriptions, inten-

tions have a more motivational character. Informally, an intention could be defined as a motivation behind any choice made during the software development process. In other words, an intention documents the "purpose" and justifies why a developer did something in a certain way. An important characteristic of an intention is that it typically cannot be found back in the software itself. Only the results of the decisions taken by a programmer are visible in the software. The "purpose" of "why" a programmer did something in a certain way, cannot be derived or extracted from the code. (Note that "what" the programmer did and "how" he did it, can, at least partly, be derived from the code.) What we want to do is precisely to make this kind of information explicit in the code so that it can be used, for example, to detect evolution conflicts (by checking whether intentions are invalidated upon evolution).

As with descriptions, intentions can be categorized in "what" and "how" intentions as well as "structural" and "behavioural" intentions. Again, both categorizations are more or less orthogonal.

What intentions motivate the intentions behind what the software is and does.

How intentions motivate why the software does things in a certain way.

Structural intentions motivate why the software is organised in a certain way and not in another way.

Behavioural intentions motivate why the software behaves in a certain way and not in another way.

There seems to be a strong correlation between intentions and requirements: an "intention" is always inspired by certain "requirements". For example, the functional requirements indicate what the software is intended to do. Requirements can be functional as well as non-functional and might not necessarily be known a priori. For example, it is possible that a programmer discovers some new important requirements (e.g. performance requirements) during the coding process itself, forcing him or her to make certain decisions. Note that not everything a developer does has a requirement associated with it. For example, programmers often make "opportunistic" implementation choices, which are not inspired by any requirement.

To conclude, let us return to the enumeration of section 1 to find some examples of intentions:

- Whereas some kinds of information (e.g. kinds 1 and 4) enumerated in section 1 are more descriptive, programming and design styles and conventions (kind 3) are more intentional: knowing which "rules of discourse" a developer uses explains to some extent why the software is organised or operates in a certain way. Examples of conventions in object-oriented development are: the law of Demeter and loose coupling [Lie96], avoiding bad super calls, well-formedness (i.e. avoiding "dangling" self sends) [SLMD96, Luc97], abstract superclass rule [Hür94] and so on. For example, the law of Demeter which states that the number of acquaintance classes of each method should be minimised, has a direct impact on the structure of the software and thus can be considered as a structural declarative intention.

- Another example are the "concerns" addressed by the methods in a class. When looking at a class in an object-oriented programme, some methods in the class address different concerns than other methods. (For example, a class implementing a node in a local area network, might address a concern "packet processing" to handle packets it receives through the network and a concern "packet forwarding" to forward packet to the next node in the network.) Information on which methods implement which concern and what the dependencies between the different concerns are is important intentional information that should be known to deal with evolution. For example, if this dependency structure is accidentally invalidated upon evolution, there probably is an evolution conflict. Concerns are an example of behavioural declarative intentions because they give an idea of what the software is intended to do and why (for example, a LAN node should implement packet handling and packet forwarding behaviour) without saying exactly how it should be done.

The above list only provides some examples

of what intentions could be. It needs to be investigated which other kinds of intentions are important and should be documented explicitly by the developers.

3 Software Annotations

In literature, many examples can be found of software descriptions that are made explicit by annotating the software with tags or constraints. (Examples of constraints are propagation patterns and examples of tags are the contract types of reuse contracts [Luc97] which document how classes in an object-oriented system are derived from other classes.) Regarding intentions, it needs to be investigated how they can be made explicit in the software by means of software annotations such as tags or constraints. Although intentions are at a higher conceptual level than descriptions and thus potentially provide more insight in the software, for the same reason it probably will be more difficult to make them explicit in the software. An additional problem is that annotations describing the intentions should be as simple as possible so that they can easily be used and understood by software developers, while being as formal as possible so that it is easy to reason about them and manipulate them in tools supporting software evolution. Therefore, it should be investigated which kinds of intentions can be made explicit in the software by explicit annotations and how.

The most important kinds of software annotations are *tags* and *constraints*:

Tags are strings, but with an implicit semantics associated to them.

Constraints are n-ary predicates on software artefacts. They facilitate adaptability by limiting the range of alternatives permitted and by documenting limitations and assumptions. Their goal is to maximise flexibility while imposing necessary limits [Lie95].

Most methodologies or tools provide some notion of tags and constraints. For example in the object-oriented modelling language UML [RJB97] *stereotypes* and *constraints* can be used to enhance the expressiveness of the language. Whereas constraints are written explicitly in UML diagrams, usually in a specific constraint language such as OCL[2], stereotypes correspond to tags with "hidden" constraints. Both stereotypes and constraints can be attached to any modelling element.

Every intention that can be expressed by means of a tag (with a hidden constraint) can also be expressed by an explicit constraint (and vice versa). We need to investigate whether it is better to use tags or constraints to express intentions. On the one hand, specific tags seem easier to use and understand by a developer than general constraints about (parts of) the software. On the other hand, if for every possible intention another tag is needed, maybe the homogeneous approach provided by constraints should be preferred.

Many other kinds of software annotations are imaginable, but they can all be considered a special case of tags and constraints:

Comments are tags without an associated semantics.

Hyperlinks to documentation (help files) or related parts in the software could be considered as a special kind of tags or comments.

Typographic styles are similar to tags (but instead of using a string a special style is used) and may have an associated semantics. For example, in UML it is possible to associate *syntactic conventions* (i.e. typographic styles) with user-defined stereotypes.

Types are a special kind of constraints restricting the range of values that can be associated with variables.

Pieces of code to be executed under some circumstances (e.g. default initialisation code) can be considered as a very special kind of constraints.

Based on the kinds of software artefacts to which tags and constraints are attached, at least two possible classification hierarchies can be constructed:

- A hierarchy based on the software development phase in which the annotated software artefacts occur.

[2] Object Constraint Language

- A hierarchy based on the granularity of the software artefacts. Software artefacts can be as fine-grained as methods or operations (e.g. an intention regarding an operation could be: "this operation addresses this concern") and as coarse-grained as complete application frameworks or software architectures (e.g. an intention regarding an object-oriented software architecture could be: "this software architecture respects the law of Demeter").

Classifications such as the above pose a number of research questions:

- Is it best to annotate artefacts early in the software development life-cycle or is it better to annotate artefacts produced in later phases? And is it better to annotate coarser- or finer-grained software artefacts?

- How can *annotation consistency* across levels of the classification hierarchies be maintained? (Annotation consistency of artefacts in different phases of the life-cycle as well as consistency between annotations of software artefacts at different levels of granularity.) As annotations can have a certain semantics, it is possible that the semantics of a certain annotation at one level conflicts with the semantics of another annotation at a different level in the hierarchy. How can we deal with this problem?

- Is it possible or useful to incrementally refine annotations at higher levels in a hierarchy to more specific annotations at lower levels?

Apart from the questions already mentioned above, the following important research questions need to be answered:

- How can the extra work for the developer of declaring annotations be supported by tools?

- How constraining do we want the annotations to be? We need to provide just enough information so that it is easy to understand and modify the software without putting too much restrictions so that evolution is still possible.

4 Conclusion

Although mere "descriptive" software annotations already enhance software understandability and adaptability, we believe that annotations motivating the "intentions" of software developers can enhance software evolution even more. For example, evolution conflicts often arise by breaking undocumented assumptions made by the original developers. Explicit intentions that document these assumptions could be used to detect or avoid such evolution conflicts.

Before reaching this goal, however, lots of research questions still need to be answered:

- Which kinds of intentions are important and should be made explicit in the software?

- About which kinds of software artefacts do we want to express intentions?

- How do intentions relate to requirements?

- Which tool support is needed for a developer to help him or her in expressing his or her intentions and to reason about these intentions?

- (How) can these intentions be made explicit in the software by means of annotations?

- Which kinds of annotations can be distinguished and how can annotations be classified?

- Which software evolution problems can be solved by reasoning about intentional annotations? (Examples needed.)

- How can software evolution problems be solved by reasoning about intentional annotations? (Theory needed.)

- Which other support software evolution can these intentional annotations provide and how?

5 Acknowledgments

Thanks to Patrick Steyaert, Carine Lucas and Tom Mens for discussing some of the ideas in this paper and for proofreading draft versions of the paper.

References

[CNFG96] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies and deviations in human-centered systems. 1996.

[CNGM96] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione. How to deal with deviations during process model enactment. 1996.

[Hür94] W.L. Hürsch. Should superclasses be abstract? In *ECOOP'94 Proceedings*. Springer-Verlag, 1994.

[KL92] Gregor Kiczales and John Lamping. Issues in the design and documentation of class libraries. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 435–451, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Lam93] John Lamping. Typing the specialization interface. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 201–214. ACM Press, 1993.

[Let86] 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.

[Lie95] 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.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with propagation patterns.* PWS Publishing Company, 1996.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1997.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* International Series in Computer Science, C.A.R. Hoare, Series Editor. Prentice Hall, 1988.

[Pen87] 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.

[RJB97] J. Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley, 1997.

[SE84] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[SLMD96] 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.

[vMV95] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.