Supporting Software Maintenance and Reengineering with Intentional Source-Code Views

Kim Mens and Bernard Poll

Département d'Ingénierie Informatique, Université catholique de Louvain
Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgium
E-mail: {Kim.Mens@, poll@student.}info.ucl.ac.be

Abstract. We propose the abstraction of *intentional source-code views* to codify essential information, about the architecture and implementation of a software system, that an engineer needs to better understand and maintain the system. We report on some experiments that investigate the usefulness of intentional source-code views in a variety of software maintenance, evolution and reengineering tasks, and present the results of these experiments in a pattern-like style.

Introduction

"A program that is used in a real world environment necessarily must change or become less useful in that environment." [Leh84]

Software systems are constantly being enhanced and adapted to accommodate to changes in their environment. Several studies have proven software maintenance and reengineering to account for a large part of the total software life cycle cost, making software maintainability a major commercial and economic factor to deal with in the development of a software system.

Our research hypothesis is that a lot of the problems related to software maintenance and reengineering are directly or indirectly caused by a documentation problem. Software documentation often addresses the needs of software analysts, but doesn't capture all information a software (re-)engineer might need to understand, maintain or reengineer a software system: important implementation choices that were made; the original programmers' intentions; how the code maps to higher-level architectural descriptions; interactions between methods, classes and modules; specifications of each piece of code; and so on.

A lot of this information is kept implicit in the engineers' heads. Some of this information may be recovered by analysing the code and its comments, but often it cannot be retrieved at all. In addition, browsing the code to understand its underlying intentions or to reverse engineer its effective architecture is a non-trivial and time consuming process and generally produces an incomplete mental picture of the software only. Such an incomplete understanding of the software can lead a reengineer to unnecessarily increase the software's complexity, or even to introduce undesired and erroneous inconsistencies in the system.

In [MMW02a] we introduced the abstraction of *intentional source-code views* as a way to *"increase our ability to understand, modularise and browse the source code by grouping together source-code entities that address the same concern."* We claim that this abstraction can be used to capture much of the information about a software system's architecture and implementation that an engineer needs to better maintain a software system. We are conducting some experiments to investigate how intentional source-code views can codify information that is essential to software maintainers and to document (in a pattern-like style) how they can use this information in a variety of software maintenance, evolution and reengineering tasks.

This research is a step towards developing technique and tool that, *without drastically changing the way in which developers work*, can:

- help engineers understand a software's architecture;
- contribute to keep an up-to-date software documentation;
- improve developers' and maintainers' efficiency;
- help engineers take implementation choices;
- help engineers to conform to an established architecture;
- avoid an evolving software's architecture to become unnecessarily complex;
- help developers and maintainers following coding conventions and standards;

- and so on.

Intentional Views

Before continuing, we explain the essence of the intentional view model. For an elaborate discussion of the model, see [MMW02a].

> *An* intentional source-code view *is a set of related (static) program entities (such as classes, instance variables, methods, method statements) that is specified by one or more alternative intentional descriptions (one of which is the 'default' intentional description). Each intentional description is an executable specification of the contained elements in the view. Such a description reflects the commonalities of the contained elements in the view, and as such, codifies a certain intention that is common to all these elements. We require that all alternative descriptions of a given view are 'extensionally consistent', in other words, after computation they should yield the same set of elements.*

The above definition highlights some key elements that turn intentional views into more than mere 'sets' of program entities:

Intentional. The sets are not (necessarily) defined by enumeration but are computed from a specification. This is useful when the software is modified as the sets are 'updated' automatically: it suffices to recompute the specification. Intentional descriptions are also more understandable and concise.

Declarative. The executable specifications are written in a declarative language, which makes them easy to read. This is important as they codify essential knowledge on the programmers' assumptions and intentions.

Alternative descriptions. Some descriptions are more intuitive; others are more efficient to compute. As such it is useful to specify both. Also, sometimes there are different natural ways in which to codify a view, depending on the perspective taken.

Extensional consistency. The consistency constraint between different alternative descriptions allows us to assess the correctness of the view definition, as well as the consistency of the actual source code (e.g., consistent usage of certain conventions and assumptions in the source code).

Deviations. Although not mentioned in the definition, for each alternative we can specify positive and negative 'deviations', i.e. elements that do not satisfy the specification of the alternative but that should be included, and elements that do satisfy the specification but should not be included. These deviations indicate 'exceptions to the general rule' made by programmers. They also help in defining intentional views incrementally: you can start out with a rough rule that has some exceptions and refine it later to make it more precise.

Relations. By relating intentional views we codify high-level structural knowledge about the source code.

Negative information. By using (logic) negation in our intentional descriptions we can codify negative information too (all program entities that do *not* have a certain desired property), which is often very powerful, especially for reengineering purposes.

Intentional views can help a reengineer because they allow him to ensure that the code — either before or after a reengineering step — has a certain structure or satisfies certain conventions. "Negative" views of all program entities that do not have a certain desired structure or do not satisfy a certain convention, could also be useful for reengineering purposes, as they group all entities that need to be reengineered (so that they do have the desired structure or satisfy the desired convention).

To help a reengineer define intentional views, we implemented a prototype tool called the Intentional View Brower [MPG03]. This tool supports the declaration of intentional views on top of the VisualWorks Smalltalk development environment. The Intentional View Browser also verifies extensional consistency.

Logic metaprogramming

The computational medium in which we specify our intentional source-code views is SOUL [MMW02b], a Prolog-like logic programming language. But SOUL is more than a mere logic programming language: it is a *metaprogramming* language, which enables logic reasoning about an underlying base language. In our case, this base language is the object-oriented programming language Smalltalk. In fact, SOUL was implemented entirely in Smalltalk, which made it quite easy to make it reason about Smalltalk (thanks to Smalltalk's strong reflective capabilities).

As a concrete example of an intentional source-code view (and of the capacity of SOUL to reason about its own underlying Smalltalk implementation), consider the definition of a view `soulLogicTestMethods`, which groups all methods that implement tests for SOUL logic predicates. (When implementing the SOUL logic libraries, we followed a kind of unit testing approach where every logic predicate was tested separately.) For readability purposes, we edited the logic code somewhat so that it resembles Prolog syntax more closely (except that Soul logic variables start with a question mark); we do assume that the reader is somewhat familiar with Prolog syntax.

```
view(soulLogicTestMethods,[extractedFromClasses,withSamePrefix]).

viewComment(soulLogicTestMethods,'This intentional view contains all methods that
implement tests for logic predicates.').

default(soulLogicTestMethods,withSamePrefix).
```

The above facts declare an intentional view `soulLogicTestMethods` with two alternatives `extractedFromClasses` and `withSamePrefix`, of which the latter is considered the default alternative. The alternative extractedFromClasses codifies the intention that a method is a *logic test method* if it belongs to a class that implements tests for logic predicates (which is verified by an auxiliary predicate `soulLogicTestClass`. An exception is made for private methods, which are only auxiliary methods that are used by the actual *logic test methods*.

```
intention(soulLogicTestMethods,extractedFromClasses,?MethodDefinition) :-
       soulLogicTestClass(?Class),
       classImplements(?Class,?Selector),
       not(privateMethod(?Class,?Selector)),
       methodDefinition(?Class,?Selector,?MethodDefinition).
```

Now suppose that an auxiliary method exists that was not put in the private protocol, where it belongs. In that case, we can still exclude it to keep the alternative intentional descriptions consistent. Of course, this is a temporary fix and we should require the developer in charge to fix the error as soon as possible. E.g.,

```
exclude(soulLogicTestMethods,extractedFromClasses,?MethodDefinition) :-
       methodDefinition(Soul.SoulTests.ErrorHandlingTest,inspectorClassUsedFor:,
                        ?MethodDefinition).
```

The other alternative withSamePrefix codifies the intention that all methods that implement tests for logic predicates have the same prefix 'test' and belong to a a subclass of a class LogicTests:

```
intention(soulLogicTestMethods,withSamePrefix,?MethodDefinition) :-
       hierarchy(Soul.SoulTests.LogicTests,?Class),
       classImplements(?Class,?Selector),
       startsWith(?Selector,test),
       methodDefinition(?Class,?Selector,?MethodDefinition).
```

Usage patterns

Rather than just enumerating the results of our experiments, we decided to present them in a pattern-like format, thus broadening their scope of usability. Being purpose-oriented, this kind of presentation enables an engineer to quickly identify what result he could reuse to aid in solving a maintenance or reengineering problem he is faced with.

Each of our *usage patterns* consists of a name, a purpose indicating what task the intentional view was used for, a rationale explaining why this task is a relevant one, a solution describing how exactly we can use intentional views to help in achieving that task and one or more concrete example(s) of such a solution.

Although not elaborated upon in this paper, we are planning to define a *pattern language* describing how these usage patterns coexist and interact. Such a pattern language will help an engineer to identify the set of patterns that address his concerns and to find out how to combine them to achieve his purposes.

We are currently using intentional views to maintain and reengineer two applications. At this stage, we defined six usage patterns, that each solve a relevant reengineering problem. Due to space limitations, we only show two of them, illustrated by a single example each. For more examples and patterns, see [MPG03].

Usage pattern 1: Enforcing coding conventions

Purpose. Verify the consistent use of certain coding conventions throughout the system.

Rationale. Programmers (and Smalltalk programmers in particular) use lots of coding conventions and 'best practice patterns' to codify their intentions [Bec97]. Unfortunately, consistent usage of such conventions and patterns strongly depends on the programmers' discipline, as it is difficult to verify that the conventions are actually respected throughout the software system.

Example. Suppose we want to enforce the convention that every mutator method assigns a value to the corresponding instance variable. We do this by defining an intentional view `mutatorMethods` with two alternatives. The extensional consistency constraint between the two alternatives takes care of the rest.

The first alternative codifies the Smalltalk naming convention that mutator methods always have the name of the instance variable that they modify, followed by a colon[1].

```
intention(mutatorMethods,byName,?M) :-
   mutatorMethod(?M,?).

mutatorMethod(?M,?V) :-
   instVar(?C,?V),
   equals(?N,{?V:}),
   classImplementsMethodNamed(?C,?N,?M).
```

The second alternative refines the first one with an extra clause which states that the method `?M` actually assigns some value to the variable `?V`. The predicate `methodWithAssignment` will traverse the entire method parse tree of the mutator method to search for such an assignment.

```
intention(mutatorMethods,byBody,?M) :-
   mutatorMethod(?M,?V),
   methodWithAssignment(?M,?V,?).
```

Extensional consistency of these two alternatives implies that all methods that follow the naming convention of mutator methods will actually assign the appropriate variable as well.

Solution. The extensional consistency constraint between the different alternative descriptions of an intentional view can be used to implicitly express an essential convention or assumption in the source code.

Enforcing such a convention will make the software cleaner and easier to understand, and thus easier to reengineer. We can also use the extensional consistency constraint for reengineering purposes. When the constraint is not satisfied, this implies that some entities do satisfy one alternative intentional description but not another. For example, it may be the case that we have a method with a mutator name, but which does not assign any value, or vice versa. Once we know the faulty entities, it is easy to reengineer them so that the extensional consistency constraint will be satisfied.

Usage pattern 2: Checking Design Consistency

Purpose. Verify consistency of the system's source code with a higher-level design diagram.

Rationale. Without a means of ensuring that the source code of a software system is, and remains, consistent with a higher-level design diagram, the design diagram soon becomes outdated and looses its relevance as high-level documentation of the source code.

Solution. To verify whether every class in, for example, a UML class diagram corresponds to one in the source code and vice versa, we declare one intentional view with two alternative definitions. The first alternative groups all classes that have been defined in the diagram, the other groups *all* existing classes in (the relevant part of) the implementation. Inconsistencies may arise either when adding a class to the source code without updating the diagram or when modifying the diagram without updating the code. These inconsistencies will be detected automatically when verifying extensional consistency of the intentional

[1] The expression `{?V:}` produces a string which is the concatenation of the string representation of the value contained in the logic variable `?V`, with a colon.

view. The same applies for methods, instance variables and class variables. Due to space limitations, we only show the view defined for classes. Note that the `inImplementation` alternative is based on the convention that all classes of the diagram are implemented in the same namespace, but can be adapted when another convention is used.

```
view(classesOfDiagram,[inDiagram,inImplementation]).

intention(classesOfDiagram,inDiagram,?ClassName) :-
       umlClass(Diagram,?ClassName,?,?,?).

intention(ClassesOfDiagram,inImplementation,?ClassName) :-
       namespaceForDiagram(?Namespace,Diagram),
       classNameInNamespace(?,?ClassName,?Namespace).
```

Example. We recently built a small application for computing the invoices of a mobile phone operator. The source code for that application was partially generated from a UML class diagram description. The above solution allowed us to verify easily when the design diagram was out of sync with the implementation and when either (part of) the code needed to be regenerated, or the diagram needed to be updated.

Again this pattern can be used for reengineering purposes, to reengineer the source code and or diagram so that they become consistent (in case the existential consistency check would fail).

Discussion

In spite of the high declarative and intuitive nature of intentional views, one might argue that it still requires an above-average engineer to define intentional views. Therefore, we are currently investigating how to facilitate the task of defining intentional views. One possibility is to offer a simpler, but maybe less expressive, language in which to define the intentional views (as opposed to using a full-fledged logic programming language). Another way is to add tool support that offers some predefined templates for the most common kinds of intentional views, which only need to be parameterised with some concrete details. A third solution is to offer a tool that helps us in semi-automatically extracting intentional views from the source code or from an enumerated set of elements. For example, some of our colleagues are investigating the use of inductive logic reasoning to derive the logic rules describing an intentional view from a set of examples contained in an extensional view [TBKG03].

Conclusion

Evolving, maintaining and reengineering software requires adequate documentation of its implementation. However, due to the software's constant evolution, this documentation is often absent, incomplete, or not synchronized with the implementation. We proposed intentional source-code views as an active documentation technique to alleviate this problem. Although creating such views is not a trivial task, the support they may offer to future software maintainers may very well be worth the initial investment. Because the different ways in which intentional views may aid software maintainers are documented in the form of usage patterns, a maintainer will be able to quickly identify what particular pattern is useful for the particular maintenance or reengineering task at hand.

References

[Leh84] M. M. Lehman. Program evolution. Information Processing & Management, 20(1-2):19-36, 1984.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[MMW02a] Kim Mens, Tom Mens, Michel Wermelinger. Maintaining software through intentional source-code views, SEKE '02 July 15-19.

[MMW02b] Kim Mens, Isabel Michiels, Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, November 2002.

[MPG03] Kim Mens, Bernard Poll, Sebastian Gonzalez. Using intentional source-code views to aid software maintenance. Paper submitted to ICSM'03.

[TBKG03] Tom Tourwé, Johan Brichau, Andy Kellens and Kris Gijbels. Induced intentional software views. Paper submitted to ESUG'03.