Supporting software evolution with Intentional Software Views

Kim Mens	Tom Mens[*]	Michel Wermelinger[†]
Département INGI	Programming Technology Lab	Departamento de Informática
Univ. catholique de Louvain	Vrije Universiteit Brussel	Universidade Nova de Lisboa
Louvain-la-Neuve, Belgium	Brussels, Belgium	2829-516 Caparica, Portugal
Kim.Mens@info.ucl.ac.be	Tom.Mens@vub.ac.be	mw@di.fct.unl.pt

ABSTRACT

Maintaining and evolving large software systems is hard. One underlying cause is that existing modularisation mechanisms are inadequate to handle crosscutting concerns. We propose *intentional software views* as an intuitive and lightweight means of modelling such concerns. They increase our ability to understand, modularise and browse the implementation by grouping together source-code entities that address a same concern. Alternative descriptions of the same intentional view can be provided and checked for consistency. In addition, the model supports the declaration, verification and enforcement of relations among intentional views. This facilitates software evolution by providing the ability to detect invalidation of important intentional relationships among concerns when the software is modified.

Keywords

intentional software views, crosscutting concerns, modularisation, evolution conflict detection

1. INTRODUCTION

1.1 Goal

We propose the model of *intentional software views* as a means of enhancing the limited modularization mechanisms provided by current-day programming languages. In particular we focus on how the proposed model — in addition to making it easier to understand and browse source code — can support software evolution. We only explain the general approach and some concrete examples that have already been implemented. A thorough explanation of the formal model and an experience report based on a real case study will be the subject of two forthcoming papers.

[*]Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)

[†]Supported by ATX Software SA, and by project POSI/32717/00 (Formal Approach to Software Architecture) funded by Fundação para a Ciência e Tecnologia.

1.2 Problem

Once software systems reach a certain size, the modularisation constructs provided by current programming languages fall short. As has been recognized by the AOP community [7], system-wide concerns often do not fit nicely into the chosen modularisations. They are said to *crosscut* the *dominant* modularisations [13]. These crosscutting concerns tend to emerge as obstacles when the developers want to evolve the software [1].

AOP addresses this problem by implementing crosscutting concerns as separate *aspects* and merging them together afterwards with a special-purpose compiler called *aspect weaver*^{TM}. Unfortunately, the AOP approach implies a completely new way of programming and is not mature enough yet to be used in every-day programming practice.

1.3 Proposed solution

We propose a more pragmatic complementary approach that provides a powerful and expressive software modularisation mechanism *on top of* an existing programming language. Instead of describing the dominant concern and the other concerns in separate aspect languages that are weaved afterwards, we allow the implementation to be modularised into an arbitrary number of user-defined *intentional views* that may crosscut the actual implementation structure and that may be overlapping. Each intentional view corresponds to an important concern that may be spread throughout the source code. It groups the set of source-code entities that address this concern. An intentional view is a *view* in the sense that it provides only partial information and does not have to be explicit in the actual source code.[1] It is *intentional* as it describes the common characteristic of the entities belonging to a view in an abstract and intuitive way that clearly expresses the 'intent' of the view. More specifically, we describe this intent in a declarative (Prolog-like) programming language.

In addition to defining intentional views, the model allows us to express, verify and enforce important relationships among intentional views. As such, many hidden assumptions in the source code are codified as explicit knowledge about the software system.

Using intentional views and their relationships to make software concerns explicit increases software maintainability and evolvability. First of all, they enhance software understanding because they provide important knowledge about where and how certain concerns are implemented and how they relate with other concerns. As such, intentional views and their relationships serve as a kind of active and enforceable documentation at an abstract level that is not explicitly available in the source code. Secondly, it becomes easier to manage the software because important concerns have

[1]In this sense, it is similar to a database view.

been made explicit in the intentional views, even if they are spread throughout the source code. Finally, when the software evolves, we can analyze the constraints imposed by the intentional views and their relationships to verify that no hidden assumptions have been invalidated. This verification can be done automatically because the description of views and their relationships is an executable specification in a declarative meta-programming language.

2. MOTIVATING EXAMPLE

Before presenting the general model of intentional views we give a concrete example of some views, how they are related, and how this information may help us in detecting and resolving interesting conflicts when the software evolves.

2.1 The case study

The running example of this paper is taken from an ongoing case study on the evolution of SOUL (Smalltalk Open Unification Language), a medium-sized object-oriented application ($>$ 150 classes) implemented in Smalltalk/VisualWorks.

In essence, SOUL is an interpreted logic programming language. It comes with LiCoR (Library for Code Reasoning), an associated library of logic predicates for reasoning about object-oriented source code at a high level of abstraction.

A particularly interesting aspect of the implementation is that it makes extensive use of *unit testing*, one of the essential ingredients of eXtreme Programming [3]. Ideally, for each method in the SOUL implementation there is a corresponding test method. Whenever a method is modified the developer needs to rerun the corresponding test method to make sure that the method still behaves as expected. Similarly, for every logic predicate in LiCoR there should be a corresponding test method that verifies whether the predicate fails and succeeds when expected. This is particularly important as incorrectly working predicates are important indicators of deeper problems with the SOUL implementation. It happened on several occasions that a predicate that had worked correctly in many earlier versions, suddenly gave rise to errors, typically caused by incorrect changes to, or optimizations of, the SOUL interpreter. With the unit testing approach such errors could be detected at a very early stage.

2.2 Intentional view examples

Test-suite completeness

In practice not every predicate has a corresponding test method. As most predicates were ported in bulk from an earlier version of SOUL in which no unit tests were available, the unit tests had to be added a posteriori on a predicate per predicate basis, which was a time-consuming and labour-intensive process. Nevertheless, the developers of SOUL were well aware that *completeness of the test-suite* was an important issue and they put quite some energy in attaining this goal.

Our model of intentional views helped the developers in achieving this completeness. Two intentional views played a crucial role. One view groups all LiCoR predicates, and another one groups all unit test methods. The completeness constraint was expressed as a relation between these two intentional views: for every predicate in the first view there must exist a corresponding test method in the second one. How exactly we defined these views and the relation will be illustrated in the next section.

Consistency between alternative definitions

The above example illustrates that relations among classifications can be used to express and verify important constraints and invariants on source code. But even the description of one single intentional view can already help us in expressing and verifying interesting assumptions and conventions in the source code.

For example, consider the intentional view that contains all logic predicates. Since both SOUL and LiCoR are implemented entirely in Smalltalk, the LiCoR predicates are wrapped in Smalltalk methods, so this view actually contains methods instead of predicates. There are two alternative ways of defining this view.

1. The first alternative uses a naming convention: all logic predicates are wrapped in methods of a class that belongs to a Smalltalk class category of which the name starts with the string *'SoulLogic'*.

2. The second alternative relies on the fact that all classes containing logic predicates must be descendants of the class *LogicRoot*. It is this particular class that defines a means of wrapping logic predicates in ordinary Smalltalk methods.

Both alternatives describe all software entities that are supposed to belong to the view. These alternatives are supposed to be consistent in the sense that they all describe exactly the same set of entities. This consistency constraint among the alternatives implicitly expresses an essential convention or assumption in the source code, namely that the naming convention in the first alternative must be respected by all subclasses of *LogicRoot*. Although this particular constraint is not 'crucial' in the sense that breaking it would not give rise to program errors, respecting it does make the software 'cleaner' and thus more understandable and easier to browse. In fact, the constraint gives an explicit semantics to the naming convention so that we can actually be *sure* that everything in a Smalltalk category with the correct name represents a real logic predicate.

A behavioural description of test methods

Intentional views may also codify some more behavioural information about the program. For example, we can define an intentional view that collects all 'correct' unit test methods for logic predicates. These test methods have in common that they invoke another method which is in charge of the actual unit testing of logic queries.

3. THE MODEL

Let us take a closer look at our approach to intentional views. It basically consists of two parts. The first part is a *language model* that describes the types of software entities in the host language we would like to view and the primitive implementation relationships in terms of which we can define high-level relationships among views. In the current paper, the language model is defined to reason about applications written in Smalltalk/VisualWorks. The second — and most important — part is the *intentional view model* itself. It defines the notions of intentional views and relations among views, as well as alternative intentions and which constraints can be imposed on them.

3.1 The host-language model

The purpose of a language model is to abstract away what is not relevant to define views (e.g., the parse tree of a method, or the fact that abstract methods are expressed in Smalltalk by self sends to `subclassResponsibility`). As such, the language model reflects only those language constructs that programmers want to reason about using views. For the same language, one can have many different language models. The particular language model for Smalltalk/VisualWorks that we will use is given in Figure 1.

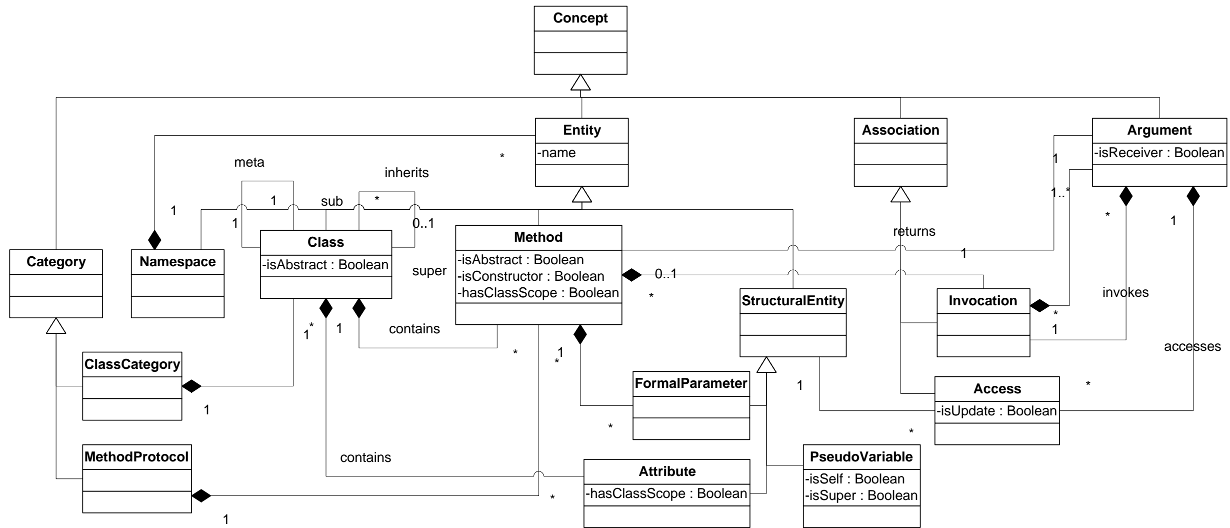

Figure 1: A particular language model for Smalltalk/VisualWorks.

To be able to reason about language entities and their relations, we use this language model to compute a set of logic predicates in a Prolog-like declarative language. For example, the software entities can be expressed using logic predicates such as: *class(?X)*, *method(?X)*, *attribute(?X)* and *category(?X)* to check that the logic variable *?X* is a class, method, variable or class category, respectively. Similarly, the relations among software entities can be expressed using logic predicates:

contains(?E,?F) expresses that entity *?E* contains entity *?F*

nameOfEntity(?E,?N) expresses that entity *?E* has name *?N*

inherits(?P,?C) expresses that class *?C* subclasses from class *?P*

inheritsTrans(?P,?C) is the transitive variant of *inherits* and expresses that *?C* belongs to the class hierarchy of *?P*

invokes(?M,?N) means that method *?M* calls a method named *?N*

accesses(?M,?A) means that method *?M* accesses attribute *?A*

The implementation of these predicates makes use of a specific meta-level interface between the logic language and Smalltalk, so that the logic predicates can directly and dynamically reason about real Smalltalk programs. The details of this meta-level interface are outside the scope of this paper; see [10] for more details.

In practice we noticed that using a full-fledged logic programming language is sometimes too expressive. In the future, we will try to restrict the expressivity by using a notation such as first-order logic, concept languages [5] or OCL [12].

3.2 The model of intentional views

3.2.1 Views

An intentional view describes a set of software entities. It contains one or more alternative intentional descriptions of this set. Each such intentional description provides an alternative insight on the intention behind the view. The description is intentional in the sense that the software entities in the view are not explicitly enumerated. Instead, Prolog-like logic predicates are used to describe all elements belonging to the view. (In the Prolog-variant we used, the keyword **if** separates the body from the head of a rule; logic variables start with question marks; a comma denotes logical conjunction; lists are delimited with <> and terms between square brackets are reified Smalltalk values.)

As an example, reconsider the intentional view from Section 2 that groups all LiCoR predicates. As explained in that section, this can be defined in two alternative ways:

view*(soulPredicates,<byCategory,byHierarchy>).*

intention*(soulPredicates,byCategory,?C) if*
class(?C), category(?K), contains(?K,?C),
nameOfEntity(?K,?N), startsWith(?N,'SoulLogic').

intention*(soulPredicates,byHierarchy,?C) if*
class(?C),
inheritsTrans([LogicRoot],?C),
not(equals(?C,[LogicRoot])).

Known exceptions or deviations to the intentional descriptions can be specified separately for each alternative. For example, the following three classes that define SOUL predicates are not captured by the *byCategory* alternative and have to be included separately:

include*(soulPredicates,byCategory,[TestClauses1]).*
include*(soulPredicates,byCategory,[TestClauses2]).*
include*(soulPredicates,byCategory,[TestClassifications]).*

But there are no exceptions that have to be excluded.

Internal consistency of the view requires that the elements in the *include* predicate are not yet present in the intention, and that elements in the *exclude* predicate must be part of the intention. Additionally, all alternatives (together with their exceptions) must be mutually consistent, i.e., they must yield exactly the same extension.

The extension can be computed automatically from the intention above by using the query *extension(soulPredicates,byCategory,?E)* which invokes the *extension/3* logic prediate. Essentially this pred-

icate computes all entities that satisfy the intentional description or that are explicitly included, whil removing all explicitly excluded entities that satisfy the intentional description.

The intuition is that the extension corresponds to the 'contents' of the view whereas the intention describes the 'meaning' of the view. One advantage of using intentional descriptions over explicit enumerations of source-code entities is that they are often much more *concise*. Another is that they are more *intuitive and precise*, as they define exactly which property all entities in a view have in common. Thirdly, intentional descriptions are more *robust towards evolution* than explicit enumerations: if the software evolves the intention will typically remain the same, but might produce another extension. Consistency among the different alternative intentional descriptions gives an indication of whether the intention has not been corrupted by the evolution. If all alternative descriptions produce the same extension there is a good chance that the intention indeed describes the intended set of entities. If not all alternatives yield the same extension set, the differences give a good indication of possible evolution conflicts.

The only disadvantage of intentional descriptions has to do with efficiency of computation. An explicit enumeration stores all values explicitly and thus can be retrieved immediately. An intentional description can be stored much more concisely, but when its values are needed, they need to be computed from the definition, which may take some time (unless a caching mechanism is used).

3.2.2 Relations

Intentional views may be explicitly related. Some of the more obvious relationships are containment (subsets), disjointness and partitions. For example, suppose that we have two more intentional views *logicPrimitives* and *logicLibraries*. Then we can express the logic fact that each SOUL predicate is either a logic primitive or belongs to a logic library, as follows:

relation*(partition,soulPredicates,*
<logicPrimitives,logicLibraries>).

As a second example, consider the view *logicTestClasses* that contains all classes that implement unit tests for SOUL predicates. We can check whether this view is consistent with the view *validPredicateTests* that describes all 'correct' unit test methods for logic predicates, by checking that each method in *validPredicateTests* belongs to a class in *logicTestClasses*, and that each of these classes contain only valid test methods. This is expressed using the following logic fact:

relation*(contains,validPredicateTests,logicTestClasses).*

where *contains* is a straightforward generalization over sets (using universal quantification) of the primitive implementation relationship *contains* between single software entities.

An example of a unary relation on views is the one that checks whether all elements in a view are of the same type, e.g.,

relation*(homogeneous(class),logicTestClasses).*
relation*(homogeneous(method),validPredicateTests).*

We can also express user-defined relations among views. For instance, the following binary relation on views expresses that there is one test entity in the second view for each entity in the first one. This generic relation, which can be used for method views as well as class views, is based on the naming convention that the name of the test entity is the name of the tested entity prepended with 'test'.

relation*(testedBy,?testedview,?testview) if*
forall(member(?e,?testedview),
exists(member(?t,?testview),
nameOfEntity(?e, ?n),
append('test', ?n, ?tn),
nameOfEntity(?t, ?tn))).

A more detailed definition should also check that the test entity actually refers to the tested entity. It was a predicate like this one that was used to check the 'test-suite completeness' constraint of Section 2.

4. DISCUSSION

We are currently conducting concrete experiments with the proposed model to understand and manage the evolution of SOUL, and to investigate how this aids maintainance of the system. We can already report some interesting results:

As the VisualWorks 3 environment did not support namespaces (a language construct similar to Java packages), many intentional views on an earlier version of SOUL were defined in terms of naming conventions (brittle). When porting SOUL to VisualWorks 5i4, some of these intentional views were codified more explicitly by using namespaces (disciplined). To ensure consistency between versions during this port, we relied on the fact that our model supports having mutually consistent alternative intentions (in our case: one using naming conventions and one using namespaces).

We already hinted on how intentional views helped the SOUL developers with their test unit approach. In addition to the examples mentioned earlier, we discovered that intentional views offer a useful abstraction for generating code (that may crosscut the implementation structure). For example, after defining an intentional view that contained all logic predicates for which *no* corresponding test method existed, we used it as a starting point to automatically generate 'stub' test methods for all those predicates. These stub test methods are very primitive and always failed when executed, so that it is easy for a software engineer to see for which predicates the test methods need to be filled in.

More experiments, including some large scale industrial ones, need to be conducted to assess how well our approach supports the effective maintenance and evolution of large software systems. Whereas intentional views clearly have some interesting benefits, they also give rise to the additional complexity that their consistency and interrelationships need to be maintained as well when the software evolves. However, since views and their relationships codify essential conceptual knowledge about a software system and its design, we believe that addressing this extra maintainance problem *explicitly* will effectively increase software quality: as illustrated earlier, inconsistencies or evolution conflicts in the views often indicate deep and important problems in the software. Obviously, to facilitate this extra maintainance task, we need to provide extra tool support. One tool we are currently working on is an intuitive user interface for browsing, editing and reasoning about views. It is also quite straightforward to provide automated tool support for checking inconsistencies among views and evolution conflicts when the software evolves: it just comes down to implementing some additional extra logic predicates.

5. RELATED WORK

Our work on intentional views builds on De Hondt's *software classification model* [4]. Software classifications are a powerful means of organizing source-code entities in a flexible and uniform manner. Similar to our software views, a *software classification* is

a collection of source-code entities, where entities can be classified in multiple classifications. Our different terminology is due to a shift in focus. De Hondt focussed on *the act of classifying* related things together. We are more interested in *the result of this act*: having different *views* on the same source-code repository.

As a special kind of software classifications, De Hondt defined a notion of *virtual* software classifications. Such classifications are not mere enumerations of source-code entities, but are computed directly from the development environment or tools. A typical example in Smalltalk are all senders or implementors of a certain method. They are 'virtual' because they do not exist as actual classifications in the environment. When changes are made to the source code, these virtual classifications are dynamically recomputed.

The research of De Hondt initiated our investigation of *intentional software views* as an intuitive and lightweight means of modeling crosscutting concerns in software. Our notion of intentional views enhances De Hondt's virtual classifications in various ways. Firstly, an intentional view can be regarded as a classification that is specified 'intentionally'. Such intentional classifications are more flexible than virtual ones, as they explicitly document which artifacts are intended to belong to the classification, instead of having them computed from the environment. Secondly, our intentional view model is more generic, since it is defined independently from a particular language model. Finally, when declared in a logic meta-programming language, the definitions of intentional views are often very intuitive and concise, and can be used in multiple ways (e.g., verificative, generative). Tourwé and De Volder [14] also explain in more detail how such a meta-programming approach supports generation of code that crosscuts the implementation structure.

Our model of intentional views bares important ressemblance with the model of *conceptual modules*, which has been used to support software reengineering tasks [2]. Like an intentional view, a conceptual module is a logical module that can be overlayed on an existing system. It is a set of lines of source code (from multiple parts of a system) that are treated as a logical unit. Our model of intentional views is more expressive and finer grained because it can contain any relevant kind of source-code entity — not only lines of source code. It is also more expressive in that a logic meta-programming language is used to reason about intentional views as opposed to a GREP-like pattern matching approach for reasoning about conceptual modules. However, this increased expressiveness comes at a cost of decreased efficiency.

Because one of the goals of our approach is to express crosscutting software concerns, it is akin to aspect-oriented programming [7]. However, we want to stress that our proposed approach is *complementary* to AOP research, as AOP technology can be used on top of intentional views, for example to generate or weave code for all artifacts that belong to a certain intentional view [14].

6. CONCLUSION

Intentional views offer a simple, intuitive and lightweight model that facilitates software understanding and maintenance. They make the code more understandable and easier to navigate through by grouping together source-code entities that address a similar concern and by allowing the definition, verification, and enforcement of relations among these groups of source-code entities. They provide an extra code structuring mechanism and as such make the software more maintainable. They allow us to ensure that certain coding conventions are consistently used throughout the source code. They provide support for software evolution by explicitly codifying hidden assumptions and constraints in the source code and by indicating which constraints have been invalidated when the software evolves. Finally, like aspects, they offer a useful abstraction for generating code that may crosscut the implementation structure.

In the future, we seek to apply our model of intentional views to architectural conformance [11, 9], architectural views [8], and refactoring [6]. As for the first two, we intend to use intentional views as a basis to specify the software architecture of a system under multiple perspectives and to check whether the implementation conforms to the architecture. As for refactoring, we hope that the crosscutting and abstract nature of intentional views might help in designing non-trivial and powerful refactoring mechanisms.

7. REFERENCES

[1] A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2002.

[2] A. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proc. Int'l Conf. Software Engineering*, pages 64–73. IEEE Computer Society Press, 1998.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[4] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Department of Computer Science, VUB, Belgium, 1998.

[5] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. *Principles of Knowledge Representation and Reasoning*, chapter Reasoning in Description Logics, pages 193–238. Studies in Logic, Language and Information. CLSI Publications, 1996.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conf. Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[8] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, November 1995.

[9] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, VUB, Belgium, October 2000.

[10] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. Software Engineering and Knowledge Engineering*, pages 236–243. Knowledge Systems Institute, 2001.

[11] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proc. of TOOLS 29 Europe 1999*, pages 33–45. IEEE Computer Society Press, 1999.

[12] OMG. *Object Constraint Language Specification*, Sept. 1997. Version 1.1, Object Management Group.

[13] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering*, 1999.

[14] T. Tourwé and K. De Volder. Using software classifications to drive code generation. Ecoop 2000 workshop on objects and classification: a natural convergence, Programming Technology Lab, VUB, Belgium, March 2000.