Maintaining software through intentional source-code views

Kim Mens
Département INGI
Univ. catholique de Louvain
Louvain-la-Neuve, Belgium
Kim.Mens@info.ucl.ac.be

Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Brussels, Belgium
Tom.Mens@vub.ac.be

Michel Wermelinger†
Departamento de Informática
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
mw@di.fct.unl.pt

ABSTRACT

Maintaining the source code of large software systems is hard. One underlying cause is that existing modularisation mechanisms are inadequate to handle crosscutting concerns. We propose *intentional source-code views* as an intuitive and lightweight means of modelling such concerns. They increase our ability to understand, modularise and browse the source code by grouping together source-code entities that address the same concern. They facilitate software development and evolution, because alternative descriptions of the same intentional view can be checked for consistency and relations among intentional views can be defined and verified. Finally, they enable us to specify knowledge developers have about source code that is not captured by traditional program documentation mechanisms.

Our intentional view model is implemented in a logic metaprogramming language that can reason about and manipulate object-oriented source code directly. The proposed model has been validated on the evolution of a medium-sized object-oriented application in Smalltalk, and a prototype tool has been implemented.

Keywords

Crosscutting concerns, modularisation, logic metaprogramming, software maintenance and evolution, validation and verification.

1. INTRODUCTION

Documenting, browsing, implementing, maintaining and evolving the source code of large software systems is hard. Once software systems reach a certain size, the modularisation constructs provided by current programming languages fall short. Typically, they support only a limited number of modularisations of the software.

As has been recognized by the aspect-oriented programming community (AOP) [6], system-wide concerns often do not fit nicely into the chosen modularisations; they *crosscut* these modularisations. Choosing another modularisation merely shifts the problem, leading to another set of concerns that crosscut the 'dominant' modularisations into which everything else needs to be fit. Perry *et al.* call this problem the *tyranny of the dominant decomposition* [14]. AOP addresses this problem by implementing crosscutting concerns as separate *aspects* and merging them afterwards with a special-purpose compiler called *aspect weaver*. Unfortunately, AOP implies a completely new way of programming and is not mature enough yet to be used in every-day programming practice.

We propose a complementary approach that provides a powerful and expressive software modularisation mechanism *on top of* an existing programming language. Instead of describing different concerns in separate aspects that are weaved afterwards, we allow the source code to be modularised into a number of user-defined *intentional views* that may crosscut the actual implementation decomposition and that may be overlapping. Each intentional view corresponds to an important (functional or non-functional) 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 characteristics 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 SOUL, a Prolog-like logic programming language that can reason about and manipulate the source code directly.

In addition to defining intentional views, our proposed model allows us to express, verify and enforce important relations 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 relations to make software concerns explicit, increases software maintainability and understandability. 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 relations serve as active and enforceable documentation at an abstract level that is not explicitly available in the source code. Secondly, it becomes easier to manage the source code because important concerns have 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 relations to verify that no assumptions have been invalidated. This

*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 Fundao para a Cincia e Tecnologia.

SEKE '02 July 15-19, Ischia, Italy

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

verification can be done automatically because the description of views and their relations is an executable description in a logic metaprogramming language.

In this paper we introduce our model of intentional views that enhances the limited modularization mechanisms of current-day programming languages. We present concrete motivating examples of intentional views in a realistic and non-trivial case study. Then we sketch the formal model behind our approach and show how it is implemented using a logic metaprogramming language. Next, we discuss some of our experiences with the model and associated prototype tool. We conclude with some related and future work.

2. CASE STUDY

The case study used in this paper to validate our approach is the development of SOUL[2] (Smalltalk Open Unification Language), a medium-sized (about 100 classes) object-oriented application implemented in VisualWorks Smalltalk. In essence, SOUL is an interpreted logic metaprogramming language. It comes with an associated library of logic predicates for reasoning at metalevel about object-oriented source code. As we will see later, SOUL is not only our case: it is also the medium in which we implemented our tool for dealing with intentional source-code views.

Before presenting our model of intentional views, we use this case study to illustrate some concrete examples of views, how they are related, and how this information may help us in understanding and maintaining source code.

2.1 Unit testing

A particularly interesting aspect of the SOUL software development process is that it makes extensive use of *unit testing*, one of the essential ingredients of eXtreme Programming [2]. This gave rise to two implicit constraints in the SOUL implementation:

C$_1$ For each method in the SOUL implementation there is a corresponding test method.

C$_2$ For every predicate in the logic library there should be a corresponding test method that verifies whether the predicate fails and succeeds when expected.

These constraints imply that whenever a method or predicate is modified the developer needs to rerun the corresponding test method, to make sure that the method or predicate still behaves as expected. Constraint C_2 is particularly important as incorrectly working predicates are important indicators of deeper problems with the SOUL language 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 many of these errors were detected at a very early stage.

In practice not every predicate has a corresponding test method. As most predicates were ported in bulk from an earlier version of SOUL without unit tests, these unit tests had to be added a posteriori on a predicate per predicate basis, which was time-consuming and labour-intensive. Our model of intentional views helped the developers in achieving *completeness of the test suite*. Two intentional views played a crucial role. One view groups all logic predicates, and another one groups all unit test methods. Completeness constraint C_2 was made explicit 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. All invalidations of this relation corresponded to a breach of constraint C_2.

Finally, we also used these views and relation to automatically generate "stub" test methods for all methods that did not have a corresponding test method. These stub test methods contained a test that always failed. As such, when running the test suites the developers' attention was triggered by the fact that a whole range of tests failed (because they still needed to be filled in). Before generating these methods the developers were not even aware of the fact certain test methods were missing unless they had manually looked for missing test methods.

[2] `http://prog.vub.ac.be/research/DMP/soul/soul2.html`

2.2 Alternative definitions of views

A second example illustrates how we can use alternative logic descriptions of the same intentional source-code view to detect inconsistencies when the software evolves. Consider the intentional view that contains all logic predicates. Since SOUL is implemented entirely in Smalltalk, the logic predicates are actually 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 *'Soul-Logic'*; (2) The second alternative relies on the fact that all classes containing logic predicates must be descendants of the same abstract class *LogicRoot*. It is this abstract class that defines a means of wrapping logic predicates in ordinary Smalltalk methods.

The fact that both alternative descriptions have to be consistent implies 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.

2.3 Meta-level interface

A third motivating example illustrates how we can use source-code views to support maintenance and evolution of source code.

The metalevel interface (MLI) of SOUL is the part of the software system that links the metalevel logic code to the base-level Smalltalk code. It has been factored out to be able to reason about different dialects of Smalltalk. More specifically, the MLI is defined as a class hierarchy, where *ExplicitMLI* is the abstract root class, and all its descendants represent different languages dialects. In an earlier version of SOUL we had the class hierarchy shown in Figure 2.

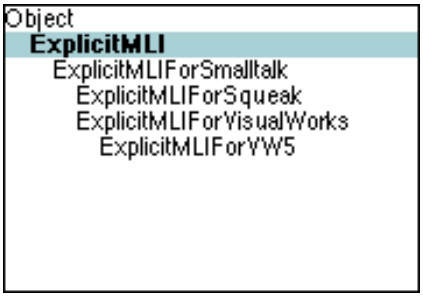

Figure 2: The SOUL *metalevel interface* class hierarchy.

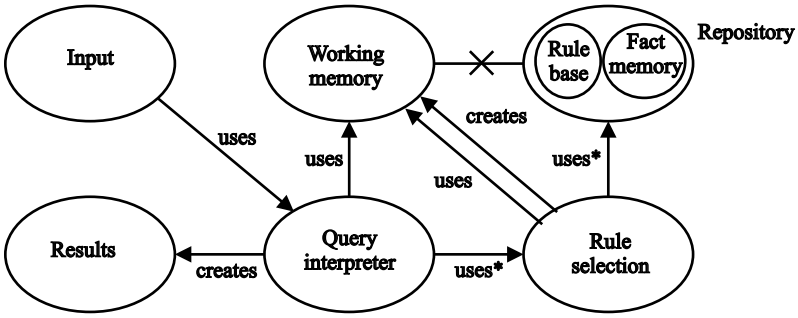

Figure 1: Rule-based architecture of SOUL

An intentional source-code view that documents this metalevel interface can be expressed in two alternative ways:
(1) All MLI classes are contained in the Smalltalk class category *'Soul-MetaLevelInterface'*;
(2) All MLI classes are descendants of the abstract class *ExplicitMLI*.

As in the previous example, the fact that both alternatives must be consistent imposes an implicit constraint on the software. Invalidations of this constraint can reveal problems during software evolution. For example, in a later version of SOUL, support for the Java programming language was added. Obviously, this required some modifications to the metalevel interface, since a new subclass *ExplicitMLIForJava* of *ExplicitMLI* needed to be added to provide an interface between Java and the metalevel logic code. These modificications gave rise to an invalidation of the above constraint, since the class *ExplicitMLIForJava* was defined in a class category 'Soul-Java' instead of 'Soul-MetaLevelInterface'. As a result, the two alternatives of the intentional view yielded different results.

Another interesting constraint expresses that each concrete metalevel interface (represented by a leaf class in the *ExplicitMLI* class hierarchy) must implement at least all predicates that are defined abstract in *ExplicitMLI*. Hence, to check whether the MLI for a specific language has been implemented completely, we simply have to check whether the view containing all abstract methods in leaf classes of *ExplicitMLI* is empty.

When verifying this view, we identified the following problematic methods for the leaf classes *ExplicitMLIForSqueak* and *ExplicitMLIForJava*.

isNameSpace:
allClassNamesInNamespace:
allNamespacesInNamespace:
allClassesInNamespace:
namespaceForClass:
namespaceAsStringFor:
allNamespaces

Additionally, for class *ExplicitMLIForJava* the method *metaclassFor:* was also identified as problematic.

These results corresponded to our intuition, since there is no direct support for namespaces in Squeak or Java. Hence, it is obvious that they do not implement methods related to namespaces. To solve the problem, all methods related to namespaces, which were originally defined in *ExplicitMLI*, were moved down to *ExplicitMLIForVisualWorks*.

The problem with *metaclassFor:* can be explained by the fact that we do not yet support metaclasses for Java. As such, *metaclassFor:* should not be defined as an abstract method in *ExplicitMLI*, but one level lower, in *ExplicitMLIForSmalltalk*.

2.4 More semantic views

We conclude with two interesting examples of views that are a bit more semantical in nature. Due to space limitations, the details of these views will not be shown in the remainder of this paper.

The first one is again related to the use of logic rules. Occasionally, a merge conflict arises when the same logic rule appears twice in the logic repository. Typically this is caused when the definition of a rule is moved from one logic module to another, but after a merge is not really removed from the old module so that it appears in both the old and the new module. This situation obviously has an impact on the behaviour of a logic program that uses this rule. In the best case, the program is much slower and produces duplicate results. In the worst case, it is just wrong. To detect this undesired situation we define a view that contains all logic rules of which there exist multiple occurences in the logic repository with *exactly the same* implementation. Once this view is defined we can use it to automatically remove all duplicate occurences of rules in that view.

A second example, which was explained in detail in an earlier paper [11], is to declaratively codify a software architecture by means of intentional views. The idea is that the architectural components correspond to intentional source-code views[3] and the architectural connectors to relations among those views. Checking conformance of the architecture to the source code then merely boils down to computing the intentional source-code views and verifying the relations among the views. Since SOUL is a logic interpreter, we considered the rule-based architecture depicted in Figure 1. The meaning of the quantifier symbols will become clear later in this paper. For more details on this particular experiment of codifying an architecture and checking its conformance to the source code we refer to [11].

3. THE LANGUAGE MODEL

The formal model of intentional views basically consists of two parts. The first part is a *language model* that describes the kinds of software entities in the host language we would like to view and the primitive implementation relations in terms of which we can define high-level relations among views. The second — and most important — part is the *intentional view model* itself, given in Section 4. It defines the notions of intentional views and relations among views, as well as which constraints can be imposed on them.

The purpose of the language model is to abstract away from implementation details that are not relevant to define views or relations among views, like the actual parse-tree representation of a method, or the way that abstract methods are represented in a par-

[3]In [11], we used the term 'virtual classification' instead of 'intentional source-code view'.

ticular language. The language model reflects only those language constructs that programmers want to reason about using views. Those concepts might correspond to actual language constructs (e.g., method calls), idiomatic conventions (e.g., naming conventions) or concepts provided by the development environment (e.g., class categories and namespaces in Smalltalk). Since the model is not supposed to be a complete and painstaking representation of the language syntax, one may have different language models for the same language, depending on the concepts of interest. In the current paper, we use intentional views to reason about applications written in VisualWorks Smalltalk, and the particular language model that we use for this purpose is shown in Figure 3. The language model is given by a restricted UML class diagram. For the moment, we only use abstract and concrete classes with public attributes, inheritance relations, aggregations and unidirectional associations. We do not make use of cardinalities in associations.

To properly define the model of intentional views, for every class, association and attribute in the UML language model, there should be a primitive operation to reason about the corresponding source-code construct. (This set of operations can be extracted automatically from the information present in the language model.) We use a logic metaprogramming approach and define these primitive operations as logic predicates in the Prolog-like language SOUL.

Below we discuss the three kinds of extracted predicates that need to be implemented. Note that logic variables start with question marks, and that the exact form of a reference to a source-code entity (e.g., method, class, variable) depends on the actual implementation language.

Class predicates. For each class α in the UML language model, there must exist a predicate α*(?entity)* that describes whether a given source-code entity is an instance of α. If the argument is uninstantiated, the predicate returns all such instances through unification. The predicate also takes into account the inheritance relations in the UML model. For example, for the language model of Figure 3, the predicate `variable(?entity)` describes that a given entity is a variable, which is either a method parameter, a class attribute, or an occurrence of a `self` pseudo variable. The same predicate can be used to find all such variables in the source code.

Attribute predicates. For each attribute α in the UML language model, we have a predicate α*(?entity, ?value)*. The predicate fails if the first argument is not an instance of a class having an attribute α. For example, the predicate `isAbstract(?entity,true)` can be used to find all Smalltalk classes and Smalltalk methods that are abstract, and `isAbstract([ExplicitMLI],?value)` should return true for `?value`.
The predicate `hasClassScope(?entity,?value)` can be used to detect whether an attribute of a Smalltalk class is either a class variable or an instance variable.

Association predicates. For each association α in the UML language model, there is a predicate α*(?from,?to)* with arguments in the same order as the association roles. For example, the predicate `inherits(?sub,?super)` checks if the first argument is a subclass of the second one, and `name(?entity,?name)` finds or checks the name of a certain entity. These predicates can also be used with one or two arguments uninstantiated.

The actual implementation of these class, attribute and association 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. Although the details of this meta-level interface are outside the scope of this paper (see [10] for more details), we show the SOUL implementation of one of the predicates. For example, there must exist a predicate `class(?entity)` that can check whether `?entity` is a valid Smalltalk class.

```
class(?entity) if
   atom(?entity),
   [ Soul.ExplicitMLI current isClass: ?entity ].
```

The above rule calls the meta-level interface for the current Smalltalk dialect to check whether the value bound to the logic variable `?entity` is an existing Smalltalk class (which is implemented by sending an appropriate method to one of the Smalltalk system classes). A second rule is needed to handle the case when the predicate is called with an unbound variable.

```
class(?entity) if
   var(?entity),
   memberOfCollection(?entity,
      [ Soul.ExplicitMLI current allClasses ]).
```

This rule calls the current meta-level interface to retrieve all possible Smalltalk classes and then unifies each of the classes in the returned collection with the logic variable `?entity`.

4. THE INTENTIONAL VIEW MODEL

While the language model defines what kinds of source-code entities and relations are of interest to the developer, the *intentional view model* defines how views and their relations can be defined for the given language.

4.1 Intentional views

An *intentional view* describes a set of source-code entities. It contains one or more alternative intentional descriptions of this set (one of which is called the 'default' intentional description). Each such description provides an alternative insight on the intention behind the view. The description is intentional in the sense that the source-code entities in the view are not explicitly enumerated. Instead, logic predicates are used to describe all entities belonging to the view. As explained in Section 3, the logic predicates are defined in terms of primitive logic predicates that correspond to the language constructs specified by the language model. In the Prolog-like logic language we use to define predicates, in addition to prefixing logic variables with question marks, the keyword **if** separates the body from the head of a rule; a comma denotes logical conjunction; lists are delimited with $<>$, and terms between square brackets are reified Smalltalk values.

Logic facts of the form `view(`*`View, Alternatives`*`)` declare the name of a view and its alternative descriptions (a list). `view-Comment(`*`View, Comment`*`)` describes the purpose of the view. The default alternative is specified by a fact of the form `default(`*`View, Alternative`*`)`. For each alternative, we must define a logic predicate `intention(`*`View,Alternative`*`,?entity)`, which describes when an entity belongs to the view according to this alternative description. Known exceptions and deviations to the intentional descriptions can be specified separately by means of facts `include(`*`View, Alternative, Entity`*`)` and `exclude(`*`View, Alternative, Entity`*`)`, that declare which entities should be included in (resp. excluded from) the description.

As an example, reconsider the intentional view from Section 2 that groups all SOUL predicates. The precise logic definition is given below. Note that for the alternative `byCategory`, we need to include three extra classes that are not captured by the category naming convention. Also note that the predicate `inHierarchy` used in the definition of the alternative `byHierarchy` is the transitive version of `inherits`.

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

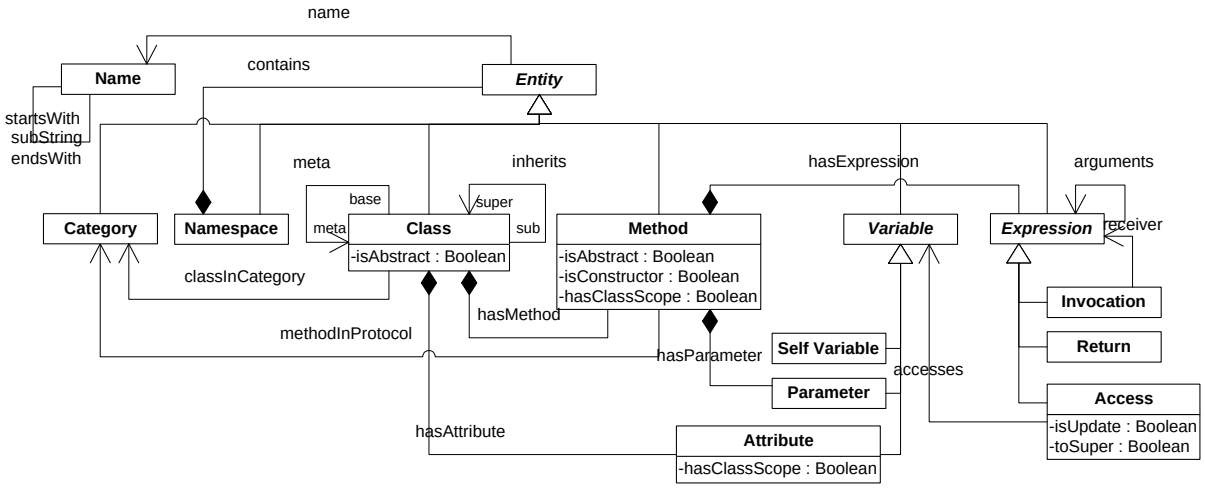

Figure 3: UML class diagram representing a particular language model for VisualWorks Smalltalk.

```
viewComment(soulPredicates,  ['This intentional
  view contains ALL classes that implement SOUL
  predicates  (= Prolog-predicates that may use
  Smalltalk code due to language symbiosis).'])

default(soulPredicates,byHierarchy).

intention(soulPredicates,byCategory,?class) if
   category(?category),
   name(?category,?catName),
   startsWith(?catName,['Soul-Logic']),
   class(?class),
   classInCategory(?class,?category).

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

intention(soulPredicates,byHierarchy,?class) if
   class(?class),
   inHierarchy(?class,[Soul.LogicRoot]).
```

Internal consistency of a view requires that the entities in an `include` predicate do *not* already satisfy the intentional description (otherwise they are already included), and that elements in an `exclude` predicate *do* satisfy the intentional description (otherwise there is no need to exclude them).

If we want to know whether a certain entity satisfies a certain intention, but no particular alternative is specified, the default alternative is assumed:

```
intention(?View,?Entity) if
    default(?View,?Alternative),
    intention(?View,?Alternative,?Entity).
```

Views can be defined in terms of other views using this last predicate. For example, the following defines the view of all SOUL metalevel interface classes for VisualWorks Smalltalk 5 only (where `soulMLI` is assumed to represent the view of all metalevel interface classes in SOUL):

```
intention(vw5MLI, byHierarchy, ?Entity) if
    extension(soulMLI, ?Entity),
    inherits(?Entity, [ExplicitMLIForVW5]).
```

The reason why we can always safely choose the default alternative is because we verify automatically that all alternative *intentional* descriptions (together with their exceptions and deviations) are mutually consistent, i.e., they yield exactly the same *extension*. The extension can be computed automatically from an intentional description by the following pre-defined predicates:

```
extension(?View,?Alt,?Entity) if
   or( intention(?View,?Alt,?Entity),
       include(?View,?Alt,?Entity) ),
   not( exclude(?View,?Alt,?Entity) ).

extension(?View, ?Entity) if
    default(?View, ?Alternative),
    extension(?View, ?Alternative, ?Entity).
```

In addition to user-defined views that can be specified as explained above, from the language model a set of primitive views is derived automatically. More precisely, for every kind of entity in the language model there is a corresponding view that contain all such source-code entities in the program under consideration. For example, there is a view `class` that contains all classes present in the Smalltalk system. Such views are defined straightforwardly as follows (where *`Entity`* is a kind of entity in the language model):

```
view(Entity,<primitive>).
viewComment(Entity, ['This intentional view
   contains all entities of kind ', Entity]).
default(Entity,primitive).
intention(Entity, primitive, ?e) if
   Entity(?e).
```

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 intentional*, as they define exactly which property all entities in a view have in common. Thirdly, intentional descriptions are more *robust towards changes* than explicit enumerations. An important 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).

4.2 Relations

Although views by themselves are already useful, they become even more useful when they can be related explicitly. Since views are *sets* of source-code entities, our model provides a series of predefined relations on sets like *empty*, *subset* and *disjoint*. For example, `subset(soulPredicates, class)` states that the intentional view *soulPredicates* includes Smalltalk classes only, because it is a subset of the primitive view *class* that groups all Smalltalk classes.

Just like a set of primitive views was derived automatically from the language model, a set of primitive relations among views can be derived automatically. More precisely, for every association in the language model with corresponding association predicate `A(?from,?to)` we can turn this into a general metapredicate between views `?V1` and `?V2` as follows:

```
A(?Q1,?V1,?Q2,?V2) if
   ?Q1( extension(?V1,?e1),
      ?Q2( extension(?V2,?e2),
         A(?e1,?e2) ) ).
```

where `?Q1` and `?Q2` represent a quantifier forall (∀), exists (∃) or existsOne (∃!). Each of these quantifiers takes two arguments: a predicate that generates values for some variable, and a formula using that variable.

For example, applying the general metapredicate above to the `inHierarchy` association predicate we could declare the fact that the view `vw5MLI` (of all SOUL MLI classes for VisualWorks Smalltalk 5) is a 'subhierarchy' of the view `soulMLI` (of *all* MLI classes in SOUL) as follows:

```
inHierarchy(forall, vw5MLI, exists, soulMLI).
```

Other examples are shown on Figure 1. For example, a `uses*` relation between methods is transformed into a relation between the views `Query interpreter` and `Rule selection` by means of two existential quantifiers (indicating that at least one method in the first view directly or indirectly uses a method in the second).

Note that there is no reason to restrict the kinds of relations between views to those that can be derived from primitive association predicates only. In fact, we can turn any predicate `A(?from,?to)` between source-code entities into a relation between views. So instead of restricting to association predicates, we allow to take any combination of class, association and attribute predicates by means of negation, conjunction and disjunction.

For example, if we define the predicate `notHasMethod` as the logical negation of `hasMethod`:

```
notHasMethod(?class, ?method) if
   not(hasMethod(?class, ?method)).
```

then we can define a relation `methodsFrom` between views that checks that all classes in the first view have *only* methods included in the second view, as follows:

```
methodsFrom(?classView, ?methodView) if
  hasMethod(forall, ?classView,
            exists, ?methodView),
  not( notHasMethod(exists, ?classView,
                    forall, ?methodView)).
```

4.3 Case study revisited

To check whether the test suites used during unit testing of SOUL are complete (as explained in Subsection 2.1), we have defined two views and a relation between them. The view *logicTestClasses* contains all classes that implement unit tests for SOUL predicates. The view *validTestMethods* describes all wellformed unit test methods for logic predicates. 'Completeness' of the test suites can be verified by checking the relation

```
methodsFrom(logicTestClasses, validTestMethods).
```

As another example that is useful for unit testing, we can define a binary relation between views that expresses that for each method in some method view there is a corresponding test method. This relation is based on the naming convention that the name of the test method is the name of the tested method prepended with 'test', and that the test method probably calls[4] the tested method. To define this relation between views we first define a relation `testedBy` between methods as follows:

```
testedBy(?m,?t) if
  name(?m, ?mn), name(?t, ?tn),
  startsWith(?tn,'test'), endsWith(?tn,?mn),
  hasExpression(?t,?e), invocation(?e),
  name(?e,?mn).
```

Once this relation has been defined (a straightforward conjunction of existing association, attribute and class predicates) we can straightforwardly declare the desired relation between views as follows (using the approach explained in the previous subsection):

```
testedBy(forall, ?methodView, exists, ?testView)
```

For more examples of semantic relations among views, we refer again to the architectural experiment of [11] that was briefly mentioned in Subsection 2.4.

5. TOOL SUPPORT

5.1 Integration with Smalltalk

An important advantage of our intentional view model is that it is *non-intrusive*. It can be added on top of an existing programming language or environment, allowing the definition of intentional views on top of the modularisation mechanisms supported by the language. As a proof of concept, we implemented a prototype tool for supporting intentional views on top of the VisualWorks development environment for the Smalltalk programming language. Figure 4 gives a schematic overview of this implementation.

The intentional view model is implemented in the logic language SOUL, which is implemented in VisualWorks Smalltalk. The library of logic predicates that is used to reason about Smalltalk code is implemented entirely in SOUL. The collection of logic predicates that implements the intentional view model uses a subset of the logic library.

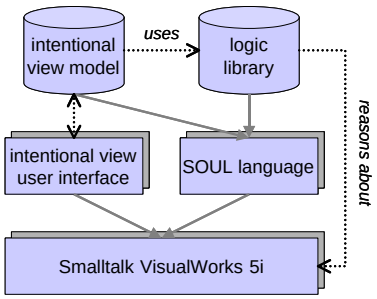

Figure 4: Experimental setup

5.2 User Interface for Intentional View Model

Because we do not want to burden a software developer with the implementation details or syntactic peculiarities that are due to the fact that the intentional view model is implemented in the logic language SOUL, we have defined an intuitive user interface.

[4]We say probably, because a static code analysis for languages with dynamic dispatching can only see which messages are sent, not which methods are actually executed.

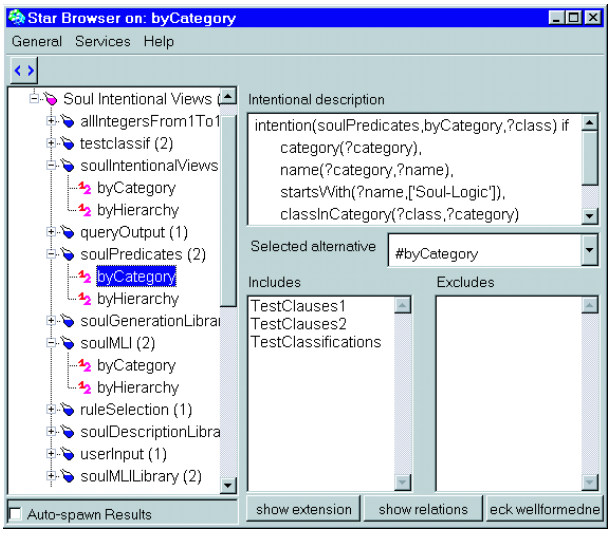

Figure 5: The Intention Editor

The *Intention Editor*, shown in Figure 5, can be used to inspect or modify alternatives for an intentional view, or to add new intentional views to the model. This requires knowledge about logic programming, as well as knowledge about the language model being used.

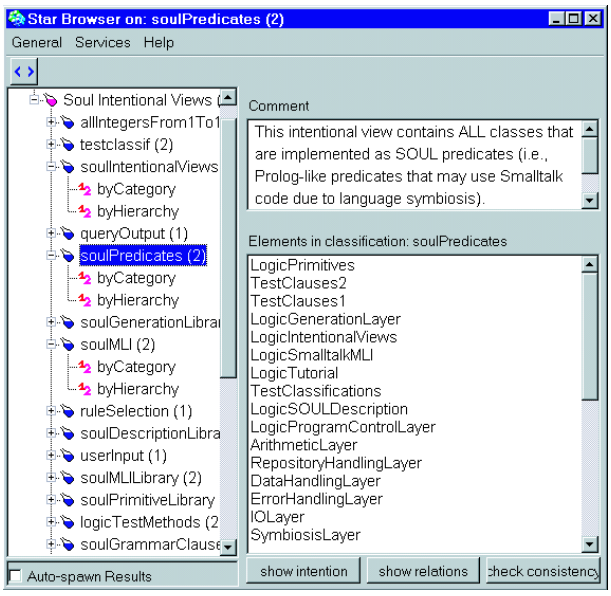

Figure 6: The Extension Viewer

The *Extension Viewer*, shown in Figure 6, allows a user to browse, for each intentional view that is defined, all source-code entities contained in this view. Each source-code entity can be selected to inspect and modify its contents (using standard Smalltalk browsers).

The Extension Viewer provides a button for checking the consistency of all alternative intentional descriptions of the current view. The Intention Editor provides a button for checking syntactic well-formedness of the currently selected alternative intentional description. From both, we can easily access the other, or open the so-called *Relation Editor*[5], which can be used to view, add, remove and modify relations between intentional views.

6. RELATED WORK

The idea of defining multiple views on software is not new and has been used in many different software engineering domains such as requirements analysis, reverse engineering and architectural recovery, re-engineering, and so on. The idea of using a logic language to reason about source code at a higher level of abstraction is not new either. What is new is our explicit focus on *intentional* source-code views that are explicitly and directly linked to the source code and the fact that the approach is tightly integrated with the software development environment. In this way, changes to the source code are directly reflected in the views and the views can be used as a starting point for generating or transforming source code.

Our work on intentional views builds on De Hondt's *software classification model* [4]. A *software classification* is a collection of source-code entities, where entities can be classified in multiple classifications. A *virtual* software classification is not a mere enumeration of source-code entities, but is computed dynamically from the development environment or tools. A typical example in Smalltalk are all senders or implementors of a certain method. Our notion of *intentional* views extends virtual classifications in various ways. Firstly, intentional views can be regarded as classifications that are specified 'intentionally'. This makes them more flexible, 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 metaprogramming language, the definitions of intentional views are often very intuitive and concise, and can be used in multiple ways (e.g., verificative, generative). We also illustrated how such a metaprogramming approach enables generation of source 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 [1]. 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. As the model of conceptual modules focusses essentially on software reengineering, it provides no support for code generation. Our approach does. Also, 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 metaprogramming 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.

Our approach is also related to earlier attempts that use logic programming to support software engineering. For example, [3] implemented a Prolog-based reverse engineering tool for analysing and querying data flow in Pascal programs. As mentioned before, an important difference with our approach is that we reason *directly* about the source code, i.e., we do not need to consult an intermediate representation (such as a database or a set of logic facts) of the source code. As such, our intentional views will always work on the most recent version of the source code. Another difference is that our logic engine is not only useful for extracting information

[5]This editor is currently under construction.

from the source code (reverse engineering) but also for generating source code (forward engineering).

Although our approach can be used to *enforce* the constraints on views by mapping them to the appropriate source-code dependencies, it falls somewhat short in doing this in an incremental manner. When small changes have been made to the source code we want to avoid having to rerun all consistency checks on the entire source. Preferably, we need a mechanism that notifies the logic environment of which changes have been made to the source code and that triggers only those rules that affect this source code. Grundy et. al. present such an approach [5]. Although we have plans to extend our logic language to support this as well, until now we have only conducted some initial experiments on this, see for example [16].

Because one of the goals of our approach is to express crosscutting software concerns, it is akin to aspect-oriented programming [6]. 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 [15].

7. CONCLUDING REMARKS

Intentional views offer a simple, intuitive and lightweight model that facilitates software understanding and maintenance. They serve as an active and enforcable documentation of the code structure. 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. The source-code entities are not grouped by enumeration but more intentionally by means of a logic predicate that specifies what it is the source-code entities have in common. These intentional views provide a generic and flexible modularisation mechanism and as such make the software more maintainable. They allow us to ensure that naming and coding conventions are consistently used throughout the source code. They support 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. They can be used for code generation. Furthermore, the verifiable intentional description of views and their relations within a logic metaprogramming approach makes it possible to provide (semi-)automated tool support for all the activities listed above.

In future work, we will refine and extend the language and view models, and the associated tool support, based on feedback gathered from further examples and case studies. Among other things, we will look at how OCL [12], which has obvious advantages at making the transition from the language model in UML to the view model, might be integrated into our logic framework. Moreover, since views and their relations capture knowledge about a software system, we are confident they will be useful to describe higher-level and more complex programming abstractions, like design patterns, architectural styles [13], and architectural views [7]. The intentional description of those abstractions might then be used not only to extract them from the source code, but also to enforce them (e.g., to check whether the implementation conforms to the software architecture [11, 9, 8]), or to help re-engineering the application.

The logic language itself also needs to be enhanced to obtain a real multiple-view software development environment (in the sense of [5]). For example, to enable a more incremental verification algorithm the language needs to be extended with a trigger or constraint-based approach that triggers only the affected rules when changes have been made to the source code. The tool should also be integrated with a version management system so that all relevant declared constraints are automatically checked when the source code is updated to a new version.

8. REFERENCES

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

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

[3] G. Canfora, A. Cimitile and U. de Carlini. A logic-based approach to reverse engineering tools production. *IEEE Transactions on Software Engineering*, 18(12), pp. 1053–1064, December 1992.

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

[5] J. Grundy, J. Hosking and W.B. Mugridge. Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering*, 24(11), pp. 960–981, November 1998.

[6] G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proc. European Conf. Object-Oriented Programming*, LNCS 1241, pp. 220–242. Springer-Verlag, 1997.

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

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

[9] K. Mens. Multiple cross-cutting architectural views. Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000).

[10] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. Software Engineering and Knowledge Engineering*, pp. 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. TOOLS 29*, pp. 33–45. IEEE Computer Society Press, 1999.

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

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[14] 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.

[15] T. Tourwé and K. De Volder. Using software classifications to drive code generation. Workshop on Objects and Classification: a Natural Convergence (ECOOP 2000).

[16] R. Wuyts. Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.