Using Intentional Source-Code Views to Aid Software Maintenance

Kim Mens Bernard Poll Sebastián González

Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgium
Fax: +32 10 45 03 45
E-mail: {Kim.Mens@,poll@student.,sgm@}info.ucl.ac.be

Abstract

The conceptual structure of existing software systems is often implicit or non-existing in the source code. We propose the lightweight abstraction of intentional source-code views *as a means of making these conceptual structures more explicit. Based on the experience gained with two case studies, we illustrate how intentional source-code views can simplify and improve software understanding, maintenance and evolution in various ways. We present the results as a catalog of usage scenarios in a pattern-like format.*

Keywords: Software maintenance, tool support, intentional source-code views, logic metaprogramming.

1 Introduction

The architecture and conceptual structure of existing software systems are often hidden or absent in the source code. Consequently, lots of the programmers' assumptions, intentions and conventions remain unrevealed, which makes it more difficult to understand, evolve and maintain the code. We propose the notion of *intentional source-code views* to codify the conceptual structure of software systems more explicitly and to provide a means of reasoning about this information to better understand and improve the code.

To investigate the practical usefulness of intentional views to aid software maintenance, we conducted two case studies. One focused on understanding and documenting an existing software system and using this documentation to check the source code for inconsistencies. In a second one we built an application from scratch to get insights on how intentional views helped in developing better systems and on how they facilitated the maintenance task.

From the experience gained with these case studies we distilled an initial catalog of typical *usage patterns*, which is summarized in table 1. Patterns 4.1 and 4.6 explain how intentional views can help in documenting and restructuring the source code to make it more understandable and more reusable. Patterns 4.2 and 4.3 show how to detect anomalies in the source code (such as coding conventions that were not respected or missing unit tests). Pattern 4.4 illustrates how intentional views can be used to verify consistency of the source code with a design diagram. Finally patterns 4.5 and 4.6 explain how intentional views can help in a code generation context (to keep track of all non-generated code or to customize the code generator).

The remainder of the paper is structured as follows. Section 2 introduces the notion of intentional source-code views. We provide a working definition, show how to define intentional source-code views in a logic metaprogramming language, give a concrete example, and present the tool we implemented for browsing and defining intentional views. Section 3 briefly presents the case studies and Section 4 discusses each of the discovered usage patterns in detail. We end the paper with a general discussion (Section 5), related work (Section 6) and some concluding remarks (Section 7).

2 Intentional views

For an elaborate discussion of the model of *intentional source-code views* (or 'intentional views', for short), we refer to [11]. Here, we explain only those details of the model that are needed to understand the examples in this paper.

2.1 Definition

Here is our working definition of the notion of 'intentional source-code view':

An intentional source-code view *is a set of related program entities (such as classes, instance variables, methods,*

Table 1. Overview of the discovered Usage Patterns

	Usage pattern	Purpose
4.1	Conceptual Structuring	Render explicit the conceptual structure of a software system
4.2	Enforcing Coding Conventions	Verify consistent usage of coding conventions throughout a system
4.3	Verifying Test-suite Completeness	Ensure full unit test coverage
4.4	Checking Design Consistency	Verify consistency of source code with a design diagram
4.5	Detecting Manual Code	In the context of code generation, retrieve all non-generated code
4.6	Software Customization	Tailor a software system to different clients

method statements)[1] *that is specified by one or more alternative descriptions (one of which is the 'default' description). Each alternative 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. In addition, 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 computational medium in which we describe the intentional views is a declarative metaprogramming language.*

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

Intentional. The sets are not 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 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). Extensional consistency is verified by our Intentional View Browser (see Subsection 2.4).

[1]For the moment, we only provide support for static program entities, although the approach could be generalized to dynamic program entities (such as class instances) as well.

Deviations. Although not mentioned in the definition, for each alternative we can also 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. For example (see Subsection 4.3), in one of our cases we wanted to express that the test-suite for an application was complete. We codified this using two views, one containing the application methods and one containing the test methods, and a relation expressing that for every method in the first view there should exist a corresponding test method in the second view.

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 property), which is often very powerful.

2.2 Logic metaprogramming

We implemented our intentional view model as well as a user-friendly *Intentional View Browser* in the *logic metaprogramming* language *Soul*. *Soul* is a dialect of the interpreted logic programming language Prolog, that is integrated in the *VisualWorks Smalltalk* development environment. It is a *meta*-programming language because it comes with a predefined library of logic predicates for meta-level reasoning about *Smalltalk* programs. Here is one example of a logic predicate that is about Smalltalk program entities:

```
classInCategory(?Class,?Category) if
   class(?Class),
   equals(?Category,[?Class category]).
```

This predicate declares the relationship between a class and its category[2]. It consist of a single rule that takes 2 ar-

[2]In VisualWorks Smalltalk, all classes are tagged with some 'category'.

guments `?Class` and `?Category` which can either be instantiated or left variable upon calling the predicate.[3] When the arguments are left variable, the rule produces multiple results to return all appropriate values for those variables.

The rule calls an auxiliary predicate `class(?Class)` which can be used to retrieve all classes in the *Smalltalk* image, or, when `?Class` is bound to a value, to check whether a certain class exists in the *Smalltalk* image. The implementation of this predicate is given in [12].

Since *Soul* has a tight symbiosis with *Smalltalk*, we can also execute *Smalltalk* expressions during the interpretation of logic rules. The expression `[?Class category]` is an example of this.[4] When the logic interpreter encounters this expression, it jumps to *Smalltalk* to send the message `category` to the *Smalltalk* class bound to the logic variable `?Class`. The interpreter then continues the logic interpretation process with the returned result.

Overall, the rule works as follows. First we verify whether `?Class` is an existing class, or bind it to `?Class` if it is variable. Then we verify whether `?Category` is the category name of that class, or bind the category name to `?Category` if it is unbound.

2.3 Example

As a concrete example of how to declare an intentional view in the logic programming language *Soul*, suppose that we want to group all classes that belong to some application, called `gsmCase` (see Subsection 3.2). First, we declare the name of the view and a list of alternative descriptions for this view. There are two alternatives, one which groups the classes by category and one which groups them by package:

```
view(gsmCase,<byCategory,byPackage>).
```

The default alternative is specified by a separate fact. We choose as default the one that is most efficient to compute:

```
default(gsmCase,byCategory).
```

Then, we describe each alternative separately, using a predicate that states which entities belong to the view according to this alternative description. E.g., the alternative description `byCategory` is defined as follows:

```
intention(gsmCase,byCategory,?Class) if
   classInCategory(?Class,?Category),
   startsWith(?Category,['GSMCase']).
```

The expression `['GSMCase']` uses *Soul*'s symbiosis with *Smalltalk* to *reify* a *Smalltalk* string into a *Soul* value.

[3]Logic variables in *Soul* are always prefixed with a question mark, as opposed to *Prolog* where they start with a capital.

[4]Note that in *Soul* square brackets [...] denote reified *Smalltalk* expressions, as opposed to *Prolog* where they denote lists. In *Soul*, lists are delimited with <>.

Known deviations of this alternative description can be specified separately by means of facts `include/3` and `exclude/3` that declare which entities should be included in (respectively excluded from) the description. For example, the class `AdHocQueryTool` is in another category but does belong to the `gsmCase` application:

```
include(gsmCase,byCategory,
        [AdHocQueryTool]).
```

We do not need to exclude any classes.

The definition of the alternative description `byPackage` is similar and accumulates all classes that belong to the Smalltalk package `GSMCase`.

2.4 The Intentional View Browser

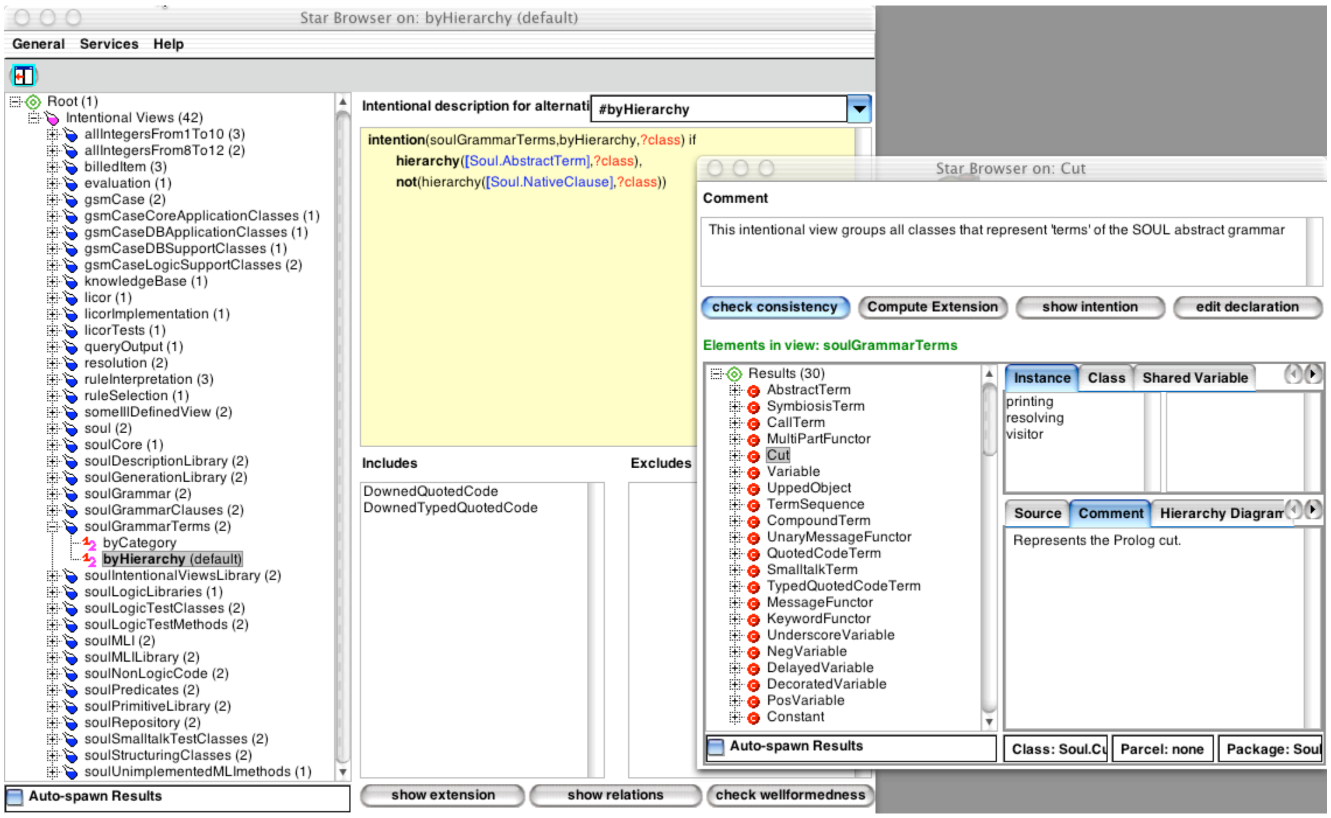
Our intentional view model is *non-intrusive*. It can be added on top of any existing programming language or environment, allowing the definition of intentional views on top of that language. We implemented a prototype tool, the *Intentional View Browser*, for supporting intentional views on top of the *VisualWorks Smalltalk* development environment. It was conceived as a plug-in for the *StarBrowser*[5], although it runs as a stand-alone application too.

The *StarBrowser* is an advanced *Smalltalk* browser that has full drag and drop support to group related program entities in sets. The *StarBrowser* also has limited support for defining 'computed' sets, where the computation is specified by a *Smalltalk* expression that returns a collection. The *Intentional View Browser* extends the functionality of the *StarBrowser* with the notion of intentional views as an advanced kind of 'computed' sets. Figure 1 shows the *Intentional View Browser* at work.

The window on the left of Figure 1 shows the *StarBrowser* with the *Intentional View Browser* plug-in. The left-most pane of this window shows all sets of related program entities that have been defined in the *StarBrowser*. Here, it only contains intentional views that have been defined with the *Intentional View Browser*. 42 intentional views have been defined, of which the names are shown, as well as (between parentheses) the number of alternative descriptions for each view. When clicking on an intentional view, the names of the alternative descriptions for that view are shown, with the default alternative distinguished (in bold). For example, the intentional view `soulGrammarTerms` has two alternatives: `byCategory` and `byHierarchy`. The default alternative is `byHierarchy`.

The right hand side of the same window can be used to browse the detailed definition of the currently selected intentional view or alternative description. In Figure 1, we selected the alternative `byHierarchy` of the view

[5]http://www.iam.unibe.ch/ ˜ wuyts/StarBrowser/



```
Root (1)
  Intentional Views (42)
    allIntegersFrom1To10 (3)
    allIntegersFrom8To12 (2)
    billedItem (3)
    evaluation (1)
    gsmCase (2)
    gsmCaseCoreApplicationClasses (1)
    gsmCaseDBApplicationClasses (1)
    gsmCaseDBSupportClasses (1)
    gsmCaseLogicSupportClasses (2)
    knowledgeBase (1)
    licor (1)
    licorImplementation (1)
    licorTests (1)
    queryOutput (1)
    resolution (2)
    ruleInterpretation (3)
    ruleSelection (1)
    someIllDefinedView (2)
    soul (2)
    soulCore (1)
    soulDescriptionLibrary (2)
    soulGenerationLibrary (2)
    soulGrammar (2)
    soulGrammarClauses (2)
    soulGrammarTerms (2)
      byCategory
      byHierarchy (default)
    soulIntentionalViewsLibrary (2)
    soulLogicLibraries (1)
    soulLogicTestClasses (2)
    soulLogicTestMethods (2)
    soulMLI (2)
    soulMLILibrary (2)
    soulNonLogicCode (2)
    soulPredicates (2)
    soulPrimitiveLibrary (2)
    soulRepository (2)
    soulSmalltalkTestClasses (2)
    soulStructuringClasses (2)
    soulUnimplementedMLImethods (1)
```

Intentional description for alternati #byHierarchy

```
intention(soulGrammarTerms,byHierarchy,?class) if
    hierarchy([Soul.AbstractTerm],?class),
    not(hierarchy([Soul.NativeClause],?class))
```

Includes

```
DownedQuotedCode
DownedTypedQuotedCode
```

Excludes

Comment

This intentional view groups all classes that represent 'terms' of the SOUL abstract grammar

Elements in view: soulGrammarTerms

```
Results (30)
  AbstractTerm
  SymbiosisTerm
  CallTerm
  MultiPartFunctor
  Cut
  Variable
  UppedObject
  TermSequence
  CompoundTerm
  UnaryMessageFunctor
  QuotedCodeTerm
  SmalltalkTerm
  TypedQuotedCodeTerm
  MessageFunctor
  KeywordFunctor
  UnderscoreVariable
  NegVariable
  DelayedVariable
  DecoratedVariable
  PosVariable
  Constant
```

```
printing
resolving
visitor
```

Represents the Prolog cut.

Figure 1. The Intentional View Browser

soulGrammarTerms and the window shows the logic predicate defining this alternative description, as well as the lists of positive and negative deviations for this alternative.

When clicking the button 'show extension', a new window is opened showing all program entities contained in this view. Each entity can be selected to inspect and modify its contents. Before computing this extension, the user is polled to check extensional consistency of the entire view first. In case the extensional consistency is not satisfied, a list containing all inconsistencies is shown to the user.

The tool also offers other features, some of which are still under development.

3 Case studies

To investigate the practical usefulness of intentional views to aid software maintenance, we conducted two case studies. The first one focused on understanding and documenting an existing software system and in the second one we built a new application from scratch.

3.1 Soul

The first case we considered (also see [11]) is the *Smalltalk* implementation of the metaprogramming language *Soul* itself. It is a medium-sized *Smalltalk* application of about 100 classes. The main goal of this case study was to try and understand the *Soul* implementation, to explicitly document its implicit conceptual structure and architecture and to codify this information in such a way that it could be used for detecting existing inconsistencies in the software as well as detecting conflicts that might be introduced upon evolution of the software.

3.2 The GSM case

The second case was a simple didactic *Smalltalk* application we built for computing invoices for a mobile phone operator. In the remainder of this paper we refer to this application as the "GSM case". The main goal of this case study was to try and understand how intentional views would facilitate and improve our software development and maintenance job.

Part of the GSM case was generated automatically from a UML class diagram using a simple code generator we built ourselves for this purpose [10]. Amongst other things, we generated the class structure, the instance creation methods and the accessor and mutator methods. The remainder of the application was written by hand.

3.3 Approach

The approach we used in both case studies was quite similar. We defined intentional views (and relations among such views), whenever we felt the need for it to help us understand, improve, (re-)structure or document the source code. To help us in defining and browsing the views, we extensively used the *Intentional View Browser*. Whenever possible, we used the specific features provided by intentional views (e.g., the 'extensional consistency' requirement or the ability to define and verify relations among views) to detect interesting inconsistencies in the code and to detect conflicts when the software evolved.

4 Usage patterns

Rather than chronologically enumerating the intentional views we defined in each of the case studies and, for each of them, explaining how they helped the software developer or maintainer, we summarise the results of our case studies under the form of "usage patterns". Every pattern 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 concrete *example* taken from one of our cases, a *solution* describing in general how exactly we can use intentional views to help achieving the task at hand and optionally some more *examples* and *related patterns*. Table 1 gives a brief overview of the discovered usage patterns.

4.1 Conceptual Structuring

Purpose: *Render explicit the "hidden" or implicit structure of an existing system.*

Rationale: When developing, maintaining or just trying to understand the source code of a software system we often discover places in the source code where the structure could be improved to make it more modular, more comprehensible or easier to browse through. Though conceptually we might have a good idea of what the intended structure should be like, often this structure is only implicit or even absent in the source code.

Example: The code of our GSM case consists of different kinds of classes: classes that were *generated*, classes for *generating* these classes and *manually-added* classes. In addition to these, there are specific classes for connecting to an external *database* and even a class that implements a little *database tool* that was used during development only. In an initial version of the GSM case, all these classes belonged to a single package without any syntactic distinction (not even in category name or in naming convention). Obviously, when the application attained a certain size, this flat structure became quite confusing, so we decided to introduce an explicit distinction by creating five intentional views (one for each type of class):

```
gsmCaseGeneratedApplClasses
gsmCaseLogicSupportClasses
gsmCaseManualApplClasses
gsmCaseDBApplClasses
gsmCaseDBSupportClasses
```

In addition, we added a constraint that these views form a partition of the entire system:

```
relation(partition, gsmCase,
  < gsmCaseGeneratedApplClasses,
    gsmCaseLogicSupportClasses,
    gsmCaseManualApplClasses,
    gsmCaseDBApplClasses,
    gsmCaseDBSupportClasses > ).
```

where `gsmCase` is the intentional view which we gave as an example in Subsection 2.3.

But while declaring these views, we were faced with the problem that we had not enough information yet to declare them intentionally due to the lack of a syntactic distinction among them. So we were forced to introduce an explicit difference between these classes at the code level first: we decided to assign them a different category name indicating which type of classes they represented. In terms of this extra information we could easily define the intentional views, for example:

```
intention(gsmCaseDBApplClasses,
          byCategory,?Class) if
  classInCategory(?Class,
              [#'GSMCase DB stuff']).
```

Without this extra information, we could only have enumerated all classes in the view, which we wanted to avoid. So, the mere fact of wanting to codify the conceptual structure with an intentional view induced a restructuring (in this case: refining the category names[6]) that made this structure more explicit in the code.

Solution: To render explicit the intended conceptual structure of a software system, we codify it by means of intentional views and relationships among them. When

[6]In Java, there are no class categories. There we could use structured class comments or refine the file hierarchy wherein the source files reside.

encoding these views is impossible because the intended structure is not present in the source code, we may first need to modify the source code slightly so that the required intentional views can be defined more easily (e.g., by tagging or commenting certain entities, by using more consistent naming or coding conventions, by refining the directory structure, . . .). After that we define the views and declare the relationships. In other words, we use the intentional views to 'overlay' a kind of *conceptual structure* on the source code. It is a 'conceptual' structure in the sense that conceptually we (want to) think of the source code as being structured like that, but in fact this structure is not yet explicit in the source code or does not even exist in the source code. However, the mere fact of defining this 'conceptual structure' explicitly as an intentional view often induces this structure explicitly in the code as well, thus enhancing the structure of the system.

Related patterns: After applying this pattern, consider applying "Enforcing coding conventions".

4.2 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 [3]. Unfortunately, consistent usage of such conventions and patterns strongly depends on the software developers' discipline as it is difficult to verify that the conventions are actually respected throughout the system.

Example: Returning to the example of usage pattern 4.1, in a later version of the GSM case we made the introduced conceptual structure more explicit by creating five separate packages, one for each set of classes. At the same time, for each of the intentional views mentioned in the example of usage pattern 4.1 we added an alternative description stating that all classes in this view belonged to a certain package. E.g.,

```
intention(gsmCaseDBApplClasses,
          byPackage,?Class) if
  classInPackage(?Class,
                ['GSMCaseDBStuff']).
```

Extensional consistency of this alternative with the already existing alternative implicitly expressed the constraint that every class in a certain package should have a certain (corresponding) category name.

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.

Example 2: A second example is taken from the *Soul* application. Consider an intentional view that contains all logic predicates. There are two alternative ways of defining this view. The first one uses a naming convention: all logic predicates are wrapped in methods of a class that belongs to a category starting with the string '*Soul-Logic*'. The second alternative states that all classes containing logic predicates are descendants of the same class *LogicRoot*, which defines a means of wrapping logic predicates in ordinary Smalltalk methods. For the details of this example we refer to [11].

Extensional consistency of both alternative descriptions implies that the naming convention specified by the first alternative must be respected by all subclasses of *LogicRoot*. Although this particular constraint is not 'crucial' in the sense that breaking it will not give rise to program errors, respecting it does make the source code '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.

Example 3: Both previous examples were quite syntactic in nature. As a more semantic example, suppose we want to enforce the convention that every mutator method assigns a value to the corresponding instance variable. Again we do this by defining an intentional view `mutatorMethods` with two alternatives. Extensional consistency takes care of the rest.

The first alternative codifies the *Smalltalk* naming convention that mutator methods are named after the instance variable they modify, followed by a colon[7].

```
intention(mutatorMethods,byName,?M) if
   mutatorMethod(?M,?).

mutatorMethod(?M,?V) if
   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.

[7]The expression `[?V,':']` is a meta-level call to Smalltalk to concatenate a colon to the variable name contained in `?V`, using the Smalltalk string concatenation operator ','.

```
intention(mutatorMethods,byBody,?M) if
   mutatorMethod(?M,?V),
   methodWithAssignment(?M,?V,?)
```

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

4.3 Verifying test-suite completeness

Purpose: *Ensure full unit test coverage.*

Rationale: When using a unit testing approach, we want the tests to cover the application as much as possible.

Example: In the *Soul* application, we wanted to ensure that every predefined logic predicate had a corresponding test method that tests correctness of the predicate. With a broad unit test coverage many implementation errors could be detected at a very early stage. Unfortunately, not every predicate had a corresponding test method, since most predicates were ported in bulk from an earlier version of *Soul* without unit tests.

Our model of intentional views helped the developers in achieving *test-suite completeness*. Two intentional views played a crucial role. One view grouped all logic predicates, another grouped all unit test methods. Test-suite completeness 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.

We also used these views and relation to automatically generate a "stub" test method for all predicates that did not have a corresponding test method. These stub test methods contained a test that always failed, thus triggering the developers' attention that the corresponding predicate still needed to be tested.

For the details of this example, again we refer to [11].

Solution: *Unit testing* relies on straightforward conventions that are codified (and thus verified) easily by intentional views and relations among views. For example, the name of the test class typically has the same name as the class being tested, appended with 'Test', and the name of the test method is typically the name of the method to be tested, prepended with 'test'.

4.4 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 entity (e.g., class, method) in 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 entities that have been defined in the diagram, the other groups *all* existing entities in the implementation. Of course, this requires that the diagram is somehow accessible by our logic metaprogramming environment. Inconsistencies may arise either when adding, for example, a class or method to the code without updating the diagram or when modifying the diagram without updating the code. These inconsistencies are detected automatically when verifying extensional consistency of the intentional view.

Example: In our GSM case, source code was partially generated from a UML class diagram. The diagram was represented by a set of logic facts, extracted using XMI from a UML class diagram in an XML-compliant drawing tool. 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 (using XMI to export the logic facts describing the diagram back to the drawing tool).

4.5 Detecting Manual Code

Purpose: *In the context of a code generation approach, retrieve all code that was not generated.*

Rationale: When we have a code generator it is very easy to regenerate the code. But care should be taken that code that was added manually afterwards does not get lost upon regenerating.

Example: As part of the code generator for our GSM case, we implemented a simple routine for backing up the manually-added code after a code generation step. Basically, this routine just computes the extension of the intentional view described in Figure 2, which contains all application classes that have not been generated.

Solution: The manual code can be defined as an intentional view which is the difference of the intentional view containing all code for the application with an intentional view containing all generated code.

```
intention(gsmCaseManualApplClasses,byDifference,?Class) if
     extension(gsmCase,?Class),
     not(extension(gsmCaseGeneratedApplClasses,?Class)).
```

or alternatively

```
intention(gsmCaseManualApplClasses,byDifference,?Class) if
      extension(difference(gsmCase,gsmCaseGeneratedApplClasses),?Class)
```

Figure 2. Intentional view containing all manually-defined classes.

Related pattern: This pattern could be generalised into one that "computes the complement[8] of another view". This is not only useful in the context of code generation, but also when collaborating with other developers. For example, suppose it is another developer's job to do a big refactoring of the user interface. When doing my own job it might be a good idea to consider all code except for the user interface code.

4.6 Software customization

Purpose: *Tailor a software system to different clients.*

Rationale: For some clients some features of a software system may be irrelevant and do not need to be included. Some other features (like debugging and unit testing) are only relevant to developers and are irrelevant for all clients. How can you easily keep track of all these different possible customizations?

Example: For debugging purposes, the code generator of our GSM case inserted a print statement in many methods throughout the application. Just dropping the code generation method responsible for this from the generator sufficed to rebuild exactly the same application, but which did not print all the debug statements upon execution. This assumes that the code generator is implemented more or less as shown in Figure 3.

Solution: Create a specific intentional view that contains all code for a particular client. It may be a good idea to create some views per functionality first from which the different users can choose which functionalities they (do not) want to include in their version.

5 Discussion and Future Work

The notion of intentional source-code views[9] was originally conceived as a means of declaring architectural knowledge explicitly at a sufficiently abstract level, while retaining the ability to perform automated consistency checking of the source code with the architecture [9]. Probably due to the limited size and duration of the conducted case studies, we did not yet find sufficient evidence to support the claim that intentional views are an ideal abstraction for that. We still believe, though, that intentional views may prove useful for describing software architectures, but more experiments are needed to validate this.

The results *do* seem to indicate a broader relevance of intentional views than we first expected, to facilitate a variety software maintenance tasks, like:

- *Documenting and restructuring* the source code to make it more understandable and more reusable (e.g., by making the conceptual structure of an existing system more explicit, by enforcing the consistent usage of naming and coding conventions throughout the system or by ensuring test-suite completeness);
- *Detecting anomalies* in the source code (e.g., non-respected coding conventions or missing unit tests);
- *Verifying consistency* of the system's source code *with a higher-level diagram* (e.g., a UML class diagram);
- *Customising* the software system for different clients;
- *Generating code* (e.g., keeping track of all non-generated code or customizing the code generator).

These positive results will lead us to further explore the usage of our intentional view model and browser as an advanced software development and maintenance tool. Related to this, we will try to achieve a more seamless integration of the *Intentional View Browser* in the *StarBrowser* and in the *VisualWorks* development environment.

Although both case studies considered in this paper were written in *VisualWorks Smalltalk*, one of the main advantages of our intentional view model is that it is generalisable to any other language (or language dialect). The only thing that is needed is a logic metaprogramming language that can reason about source code for that particular language. The logic metaprogramming language does not even have

[8] with respect to all entities of a certain kind that are currently known to the development environment

[9] In [9] and earlier papers we used the term 'virtual classification' instead of 'intentional source-code view'.

```
generateCode if
   forall( intention(generationForClient,?GenerationPredicate),
           call(?GenerationPredicate) ).
```

where the intentional view `generationForClient` is defined as

```
intention(generationForClient,default,?GenerationPredicate) if
      extension(generationPredicates,?GenerationPredicate),
      not(predicateName(?GenerationPredicate,generateDebugStatements)).
```

Figure 3. Customisation of our code generator for a particular client.

to be written in the language it reasons about. For example, Johan Fabry is working on *SoulJava* [4], a variant of *Soul*, still written in *Smalltalk*, but which supports reasoning about and manipulating *Java* source code. Related to this, we are also actively researching how the *Soul* environment can be turned into a language-independent framework for reasoning about source code. Another thing we are envisioning is the development of an Intentional View Browser for Java as an Eclipse plug-in.

In spite of the high declarative and intuitive nature of intentional views, one might argue that it still requires an above-average developer 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 which offers some predefined templates for the most common kinds of intentional views, that 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, Tourwé et. al. investigate the use of inductive logic reasoning to derive the logic rules describing an intentional view from a set of examples contained in an extensionally-defined view [17].

6 Related work

Conceptual modules [2, 1] bear close resemblance to intentional source-code views, both in spirit and in approach. Like an intentional view, a conceptual module is a logical module that can be overlaid 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 opposed to our approach, conceptual module provides no support for code generation. Our approach seems more expressive and finer grained too because it is not restricted to lines of source code only. It is also more expressive in that a logic metaprogramming language is used to reason about intentional views as opposed to a regular expression pattern matching approach. But this increased expressiveness comes at a cost of decreased efficiency. (Though nothing prohibits us from implementing in our logic metalanguage a highly-optimized predicate for doing string-pattern matching and using that predicate whenever efficiency is crucial.)

Concern graphs [14] are an abstraction meant to localize software concerns by means of graph representations of program entities (nodes) and their relations (edges). Again, concern graphs differ from intentional views in the granularity of the representation. The number of representable entities is deliberatly kept to a minimum (essentially: classes, methods and attributes) and there is a fixed set of 7 kinds of relations among the entities (“declares”, “calls”, “reads”, “writes”, ...). This deliberate restriction imposed by the concern graph approach makes the expressions of concerns concise. Intentional source-code views do not impose such a restriction and are therefore more expressive. Thanks to the strong symbiosis with *Smalltalk*, they can reason about anything *Smalltalk* can reason about. But again sometimes this may make the expressions somewhat less concise or less efficient.

Many other approaches exist where multiple views on the same software system are offered, e.g. [6]. There are even some approaches that use a logic programming language for this. However, due to the large variety of existing approaches it is not clear to us yet where exactly our approach fits in and what its advantages and disadvantages over these other approaches are.

There exist interesting relations with database research too. At the risk of oversimplifying things, if we would consider the source-code repository as a mere database of source-code entities, one could say that intentional source-code views are similar to *database views*. In fact [6] mentions how “*database views [...] can be used to build multiple-view systems where the views are informed of changes to model objects and requery the model to update the view's state. [...] Unidirectional constraint systems [...] use constraint rules between software specification components which, when triggered, automatically update affected structures or flag the presence of inconsistencies.*”

Combined with a code-generation approach, intentional views may prove useful as an enabling technology for achieving *aspect-oriented programming* [8], by generating or weaving code for all entities that belong to a certain intentional view. As such, a lot of work related to aspect-oriented programming is also related to intentional source-code views (e.g., subject-oriented programming [7], multi-dimensional separation of concerns [16], concern space modeling [15] and the AspectBrowser [5]) . A detailed investigation of how intentional views may aid aspect-oriented software development remains future work.

Finally, we repeat that work exists on using intentional views to describe software architectures [9, 13], but that more experiments are needed to further refine the model.

7 Conclusion

Whereas in [11] we presented the model of intentional source-code views, this paper reports on two case studies we conducted to investigate the practical usability of intentional views. We expected the results to show the usefulness of intentional views to codify the "architecture of existing systems". Instead the case studies showed how the specific features of intentional views — their declarative and intentional yet executable nature, the ability to define verifiable relations among views, the ability to define several alternatives of a view and the requirement of 'extensional consistency' among these alternatives — could be exploited to facilitate a whole range of software development and maintenance tasks. For ease of understanding, we expressed the results under the form of *usage patterns*. Taking into account the rather limited scope of the case studies, we are convinced that the discovered patterns constitute only the tip of the iceberg and that many more useful applications of intentional views to aid software maintenance exist.

8 Acknowledgements

We are very grateful to Johan Brichau, Roel Wuyts, Tom Mens and Michel Wermelinger for proofreading this paper. We thank Roel Wuyts for his help with building the *Intentional View Browser* and integrating it in his *StarBrowser*.

References

[1] A. Baniassad and G. Murphy. Conceptual module querying for software reengineering. In *Proceedings of ICSE1998*, pages 64–73. IEEE Computer Society Press, 1998.

[2] A. Baniassad, G. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *Proceedings of AOSD2002*, 2002.

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

[4] J. Fabry. Supporting development of Enterprise JavaBeans through declarative meta-programming. In Z. Bellahséne, D. Patel, and C. Rollan, editors, *Proceedings of OOIS 2002*, volume 2425 of *Lecture Notes in Computer Science*, pages 280–285. Springer-Verlag, 2002.

[5] W. Griswold, Y. Kato, and J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS99-0640, Department of Computer Science and Engineering, University of California, San Diego, December 1999.

[6] J. Grundy, J. Hosking, and W. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11), November 1998.

[7] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA1993*, pages 411–428. ACM Press, 1993.

[8] G. Kiczales. Aspect-oriented programming. In *Proceedings of ECOOP 1997*. Springer, 1997. Invited presentation.

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

[10] K. Mens, J. Fabry, and J. Del Valle. Multi-language code generation with logic metaprogramming — an experience report. Submitted to WCRE2003.

[11] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*. ACM, 2002.

[12] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications Applications*, 23(4):405–431, November 2002.

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

[14] M. Robillard and G. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of ICSE2002*, pages 406–416. ACM Press, 2002.

[15] S. Sutton and I. Rouvellou. Advanced separation of concerns for component evolution. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution (OOPSLA 2001)*, October 2001.

[16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE1999*, 1999.

[17] T. Tourwé, J. Brichau, A. Kellens, and K. Gijbels. Induced intentional software views. In *Proceedings of ESUG2003*, 2003.