Co-evolving Code and Design with Intensional Views — A Case Study

Kim Mens *

Département d'Ingénierie Informatique (INGI)
Université catholique de Louvain (UCL)
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Andy Kellens

Departement Informatica (DINF)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, B-1050 Brussel, Belgium

Frédéric Pluquet   Roel Wuyts

Départment d'Informatique
Université Libre de Bruxelles (ULB)
Boulevard du Triomphe - CP212, B-1050 Bruxelles, Belgium

Abstract

Intensional views and relations have been proposed as a way of actively documenting high-level structural regularities in the source code of a software system. By checking conformance of these intensional views and relations against the source code, they supposedly facilitate a variety of software maintenance and evolution tasks. In this paper, by performing a case study on three different versions of the *SmallWiki* application, we critically analyze in how far the model of intensional views and its current generation of tools provide support for co-evolving high-level design and source code of a software system.

Key words: Case study, co-evolution, intensional views and relations, *SmallWiki*.

* Corresponding author.
Email addresses: `Kim.Mens@info.ucl.ac.be` (Kim Mens), `akellens@vub.ac.be` (Andy Kellens), `fpluquet@yahoo.fr` (Frédéric Pluquet), `roel.wuyts@ulb.ac.be` (Roel Wuyts).
URLs: `http://www.info.ucl.ac.be/~km` (Kim Mens), `http://prog.vub.ac.be/~akellens/` (Andy Kellens), `http://homepages.ulb.ac.be/~rowuyts/` (Roel Wuyts).

Accepted for publication at ESUG Conference 2005 Research Track (www.esug.org)

1 Introduction

Maintaining the source code of long-lived software systems requires an adequate documentation of their intended design. However, due to their constant evolution, it is often hard to keep their source code and design synchronized. This is partly due to the fact that current-day integrative development environments still focus too much on writing code and too little on supporting maintenance and evolution tasks [1].

Intensional source-code views and relations [2,3,4,5] have been proposed as an active documentation technique that addresses some of these problems. They increase our ability to understand and document the code and its design by grouping together structurally related source-code entities. They facilitate software maintenance and evolution, because alternative descriptions of the same intensional view can be checked for consistency and because relations between intensional views can be defined and verified against the source code.

In [2] we explained how to codify software architectures by means of intensional source-code views [1] and how to check conformance of those architectures with the source code. In [3] we proposed intensional views as an intuitive and lightweight but verifiable means of documenting crosscutting concerns in a software system. In [4] we discussed how intensional views facilitate a variety of software understanding, maintenance and evolution tasks. Finally, [5] emphasized on documenting and verifying high-level *relations* between intensional views. We also discussed the analogy of testing structural source-code regularities in a software system by means of intensional views and relations with testing the behavior of a software system by means of unit tests.

To define and verify intensional views and their relations we built a tool suite which we called *IntensiVE*. This 'Intensional View Environment' was implemented entirely in and seamlessly integrated with the *VisualWorks Smalltalk* development environment and comprises, amongst others, the following tools:

The Intensional View Editor (Fig. 1) allows us to document relevant concerns in the source code in terms of intensional views and to inspect the source-code entities corresponding to such concerns.

The View Consistency Checker (Fig. 2) allows us to verify consistency between different alternative descriptions of an intensional view, with respect to the current source-code base, and to provide fine-grained feedback on the differences between these alternative definitions.

The Relation Editor (Fig. 3) allows us to document high-level relationships between intensional views, as well as known deviations of these relationships in the source code.

[1] called 'virtual software classifications' in that paper

The Relation Checker (Fig. 4) allows us to verify these relations against the current source code, and provides fine-grained feedback on their validity.

Whereas older versions of these tools have been reported on briefly in [5], we have recently re-implemented them entirely to improve their efficiency, persistence and integration with version 2 of the *StarBrowser* [6], an advanced source code browser for *VisualWorks Smalltalk*. In addition to having the logic query language *Soul* [7] as underlying language in which to describe the intensional views and relations, the tools now offer support for using *Smalltalk* too as query language to reason about source code. Another novel feature is the ability to define nested views, which allows us to create context-specific views. Finally and most importantly, we added support for visualizing intensional views and relations (see Fig. 5), by relying on *CodeCrawler* [8], a reverse engineering tool which combines software metrics and visualization.

The aim of this paper is to perform a critical evaluation of the current generation of tools, including the new opportunities offered by the visualization tool, to support co-evolution of high-level design and source-code of a medium-sized *Smalltalk* application. The case we selected for this study is *SmallWiki* [9], an object-oriented Wiki implementation in *Smalltalk*. We documented the intended design of an early version of *SmallWiki* and observed how this documentation helped us in better understanding the software and its implementation structure, as well as in discovering certain structural irregularities in its source code. Then we verified this design documentation against two more recent versions of *SmallWiki* and discovered some interesting ways in which the source code and its design evolved.

From the experiences gained with this case study, we distilled a list of lessons learned about the model of intensional views and relations and its associated tools, in particular on how they support co-evolution of source code and higher-level design. Amongst others we learned that documenting the design of a software system with intensional views and relations allowed us not only to detect interesting structural inconsistencies introduced in the code upon evolution, but also that the process of documenting itself helped us to better understand the source code and how it evolved. A dedicated visualization which highlights what views and relations have become inconsistent with the code, proved very useful since it allowed us to readily assess the impact of an evolution step and locate potential structural problems. Finally, the ability of using and combining both logic and *Smalltalk* queries had the advantage that we could always choose the query language most appropriate to our needs, that is, the one that yields the most compact and declarative queries.

2 Experimental Setup: SmallWiki

A Wiki is a collaborative web application that allows users to add content, but also allows anyone to edit the content. *SmallWiki* [9] is a fully object-oriented and extensible Wiki framework that was developed entirely in *VisualWorks Smalltalk*. As opposed to most other Wiki implementations, which are hard to adapt, SmallWiki has been designed from the start with extensibility in mind. It has a clean object-oriented design where all entities that can be stored in web pages (text, links, tables, lists) are explicitly modelled as objects. Everything in *SmallWiki* is designed to be extended: page types, storage mechanism, actions, security mechanism, web-server, etc. Plug-ins can be shared within the community and loaded independently of each other into the system.

We decided to use *SmallWiki* for our case study for several reasons. Because it is open source, its source code is freely available. Secondly, many versions exist, from very early versions up until the stable versions that are currently in use at several places. Thirdly, it is a non-trivial piece of software, yet still manageable in size and complexity. We studied the following versions of *SmallWiki*:

Version 1.54 (14-12-2002) was the first internal release of *SmallWiki*, offering an operational Wiki server with rather limited functionality: only the rendering and editing of fairly simple Wiki pages was supported. This version contained 63 classes and 424 methods.

Version 1.90 (15-01-2003) covered only one extra month of development (thus limiting the risk of having a version that was too drastically different from the first version studied). Nevertheless, this month represented quite an active period of development with several releases a day (thus making it a non-trivial version to study). This version contained 8 more classes (71 in total) but many more methods (633). An important change with respect to version 1.54 was that in this newer version the methods responsible for rendering HTML code were refactored.

Version 1.304 (16-11-2003) was chosen because it covered a larger development period (almost 1 year) with lots of intermediate versions. This allowed us to study the problem of synchronizing design documentation and source code over a longer time interval. With 108 classes and 1219 methods, this version was significantly larger than the previous two.

In order to study the usefulness of intensional views and relations to document the design structure of an evolving software system, we conducted the following experiments on the different versions:

(1) We started by codifying the design of version 1.54 and investigated how this documentation helped us in better understanding the code structure as well as some of the adopted naming and coding conventions.

(2) We then verified this structural documentation against the more recent version 1.90 and drew conclusions about how SmallWiki evolved, and about the consequences of this evolution on the documented structure.
(3) Finally, we verified the documentation against the most recent version studied (1.304) and observed that the design remained relatively stable, even after this longer development period.

3 *IntensiVE*

Before describing our experiments in more detail, in this section we give an overview of the model of Intensional Views and Relations, together with its associated tool suite: *IntensiVE*, or Intensional View Environment. The following five subsections each focus on one of the major sub-tools of the environment namely the intensional view editor, the view consistency checker, the relation editor, the relation checker, and the intensional view displayer. Along the way we explain the underlying model of intensional views and relations.

3.1 The Intensional View Editor

An *Intensional View* is a set of source-code entities (classes or methods) which are structurally similar. Instead of enumerating all elements that make up a view, it is defined by means of an *intension*: an executable description which yields, upon execution, the set of entities belonging to the view, also called the *extension* of the view.

The *Intensional View Editor* (Fig. 1) is our main tool for creating and manipulating views. On the screenshot, the left pane shows all defined views in a tree representation. The right hand side shows the Intensional View Editor opened on a view named 'Execute Methods'. This view groups all methods responsible for executing actions on Wiki pages. Since all these methods are classified in an 'action' method protocol, we provide the following intension for the Execute Methods view: `methodInProtocol(?entity,action)`. This query, written in the logic language *Soul* [7], binds occurrences of methods in the 'action' protocol to the free logic variable `?entity`. By convention, the Intensional View Editor assumes that a logic query has a free variable named `?entity` and calculates the view extension as the accumulation of all bindings to that variable. When using *Smalltalk* as query language, it suffices to write a *Smalltalk* block that returns a collection. E.g., we can define a view of all *SmallWiki* classes by means of a *Smalltalk* expression `SmallWiki allClasses`.

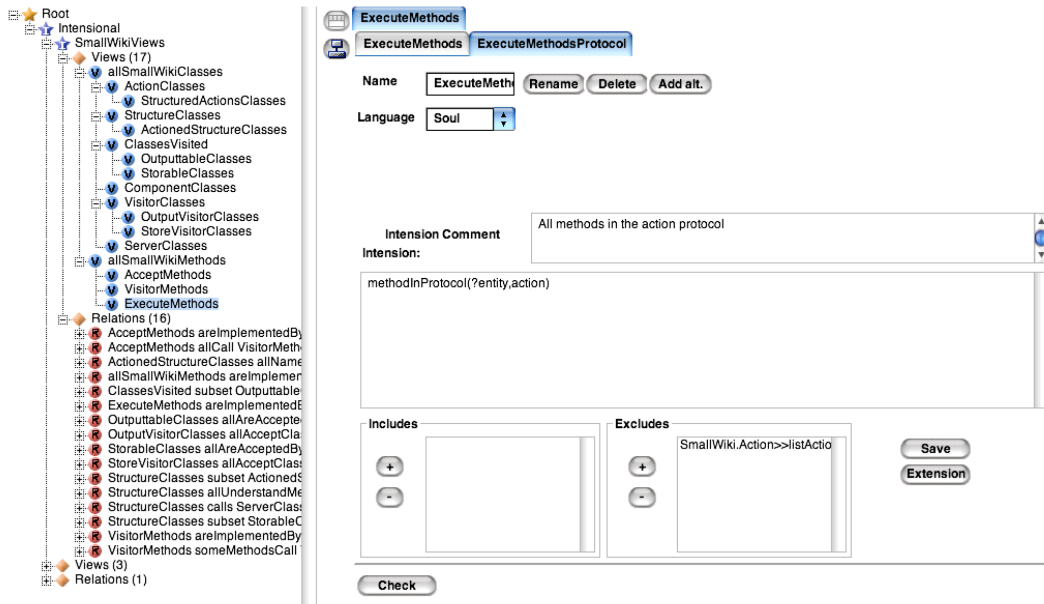

Fig. 1. The Intensional View Editor at work

Notice in the screenshot (left pane) that this view is defined as a subview of the view containing 'all SmallWiki methods'. The semantics of defining a view as subview of another one is that the intension of the subview is calculated in the context of the parent view. In other words, evaluating the intension of the Execute Methods view results in all methods which belong to the extension of the view 'all SmallWiki methods' but also to an `action` method protocol.

The tool also supports the explicit exclusion (resp. inclusion) of an entity from a view. For example, the method `listActions`, implemented on the `Action` class, is part of the computed extension of the Execute Methods view, but is not really an execute method. Hence we explicitly excluded it from the view, by putting it in the 'excludes set' of the view. Analogously, we have an 'includes set' of entities that should be included in a view, even though they do not satisfy the intension.

Intensional Views allow the definition of multiple alternative descriptions for the same view. This ability, together with the requirement of extensionally consistency (explained in the next subsection), provides an elegant way of declaring interesting naming and coding conventions to be respected by the entities of a view, as we will see in Section 4.

3.2 The View Consistency Checker

Fig. 2 shows the *View Consistency Checker*. This tool is used to verify that the different alternative descriptions of a same view are *extensionally consistent*,

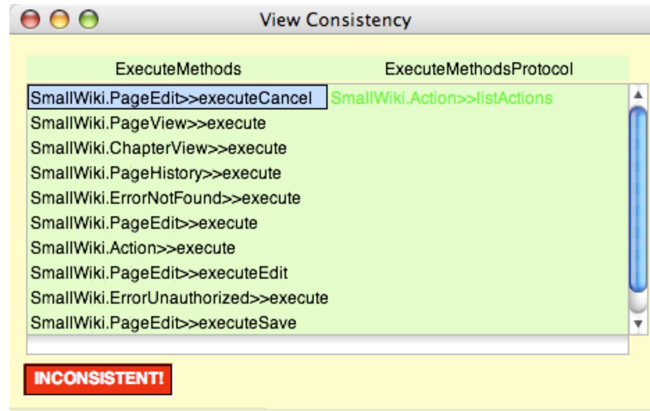

Fig. 2. The View Consistency Checker at work

meaning that they all produce the same extension. When this constraint is violated, the tool provides appropriate feedback on what entities are in cause.

To illustrate this consider the Execute Methods view again. In addition to the intension already described above, we defined an alternative description based on the observation that the names of all execute methods start with the string 'execute'. Fig. 2 shows the result of checking extensional consistency between these two alternatives of the Execute Methods view. Note that we checked extensional consistency before having explicitly excluded `listActions` from the second alternative of the view. In fact, it was precisely the feedback from the View Consistency Checker that motivated us to take a look at the implementation of that method and decide that it was a deviating case.

The tool shows the user a column per alternative description of the view. The first column contains the extension of the main alternative (by default this is the first alternative of the view, but double-clicking a column changes the main alternative); the other columns contain the delta between the extension of the main alternative and the alternative represented by the column. If an element does not exist in the main alternative, it is coloured green. Elements present in the main alternative, but not in the other are displayed in red.

3.3 The Relation Editor

The *Relation Editor* allows a user to document relations between intensional views. Our model currently supports only relations of the canonical form:

$$\mathcal{Q}_1\ x \in Source : \mathcal{Q}_2\ y \in Target : x\ R\ y$$

where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are either logic quantifiers $\forall$, $\exists$, $\exists!$, $\nexists$ or more fuzzy quanti-

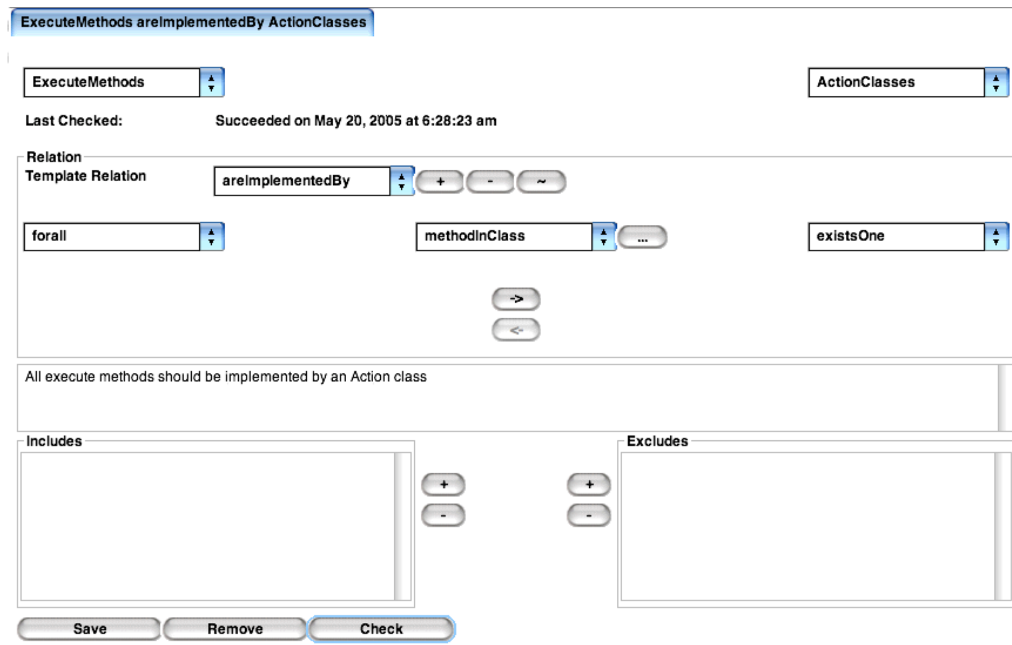

Fig. 3. The Relation Editor at work

fiers[2] like *some*, *few*, *many* or *most*. *Source* and *Target* represent intensional views and R is a binary predicate over the source-code entities (denoted by x and y) contained in those views. A simple example of an intensional relation is that all Execute Methods are implemented by an Action Class (we define this view in Section 4.1). Fig. 3 shows the Relation Editor opened on this relation. Expressed in the canonical form above, the relation was defined as:

$$\forall\ x \in ExecuteMethods : \exists!\ y \in ActionClasses : x\ methodInClass\ y$$

To define a binary predicate R over source-code entities, in terms of which intensional relations can be defined, our tool offers two possibilities. In addition to defining the predicate directly in *Smalltalk* (using a *Smalltalk* block that takes two arguments and returns a boolean), the user can opt to use a *Soul* predicate (typically using LiCoR, an extensive library of *Soul* predicates to reason about source code). For concrete examples we refer to Subsection 4.2.

Like the Intensional View Editor, the Relation Editor supports the explicit declaration of deviating cases. It allows a user to specify explicitly tuples of source-code entities to be included in or excluded from the relation.

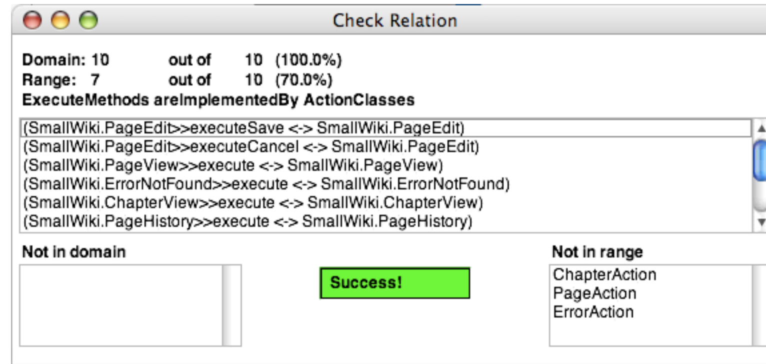

Fig. 4. The Relation Checker at work

3.4 The Relation Checker

When pressing the 'Check' button in the Relation Editor (Fig. 3), the validity of a relation with respect to the source code is checked and the user is presented an instance of the *Relation Checker* (Fig. 4). Besides reporting whether the relation holds, the tool presents the user a list of all tuples for which the relation is valid as well as some statistics on how many elements from source and target participate in the relation. It also lists all entities from the source view which are *not in the domain* of the relation as well as all entities in the target which are not reached by the relation. When a relation does not succeed, a user can use this information to determine for which source code entities the documented relation and the source code are no longer synchronized.

3.5 The Intensional View Displayer

All tools above support a user in manipulating (declaring, modifying, renaming, removing, verifying and saving) intensional views and relations. What is still missing is a visualization tool that provides a user with a global and compact drawing of all defined views and relations (or a relevant subset thereof). This is the purpose of the *Intensional View Displayer* depicted in Fig. 5. For a given selection of views, the displayer shows all these views, all their alternative descriptions, all subview links and all intensional relations in which those views take part. The views are laid out automatically in a hierarchy that reflects the view nesting, but the layout can be modified and stored manually.

Since the visualization tool is defined on top of *CodeCrawler* [8], a reverse engineering tool which combines software metrics and visualization, by making intelligent use of metrics we can highlight important characteristics of inten-

[2] The fuzzy quantifiers are defined in terms of a minimum or maximum number of elements for which the condition should hold.

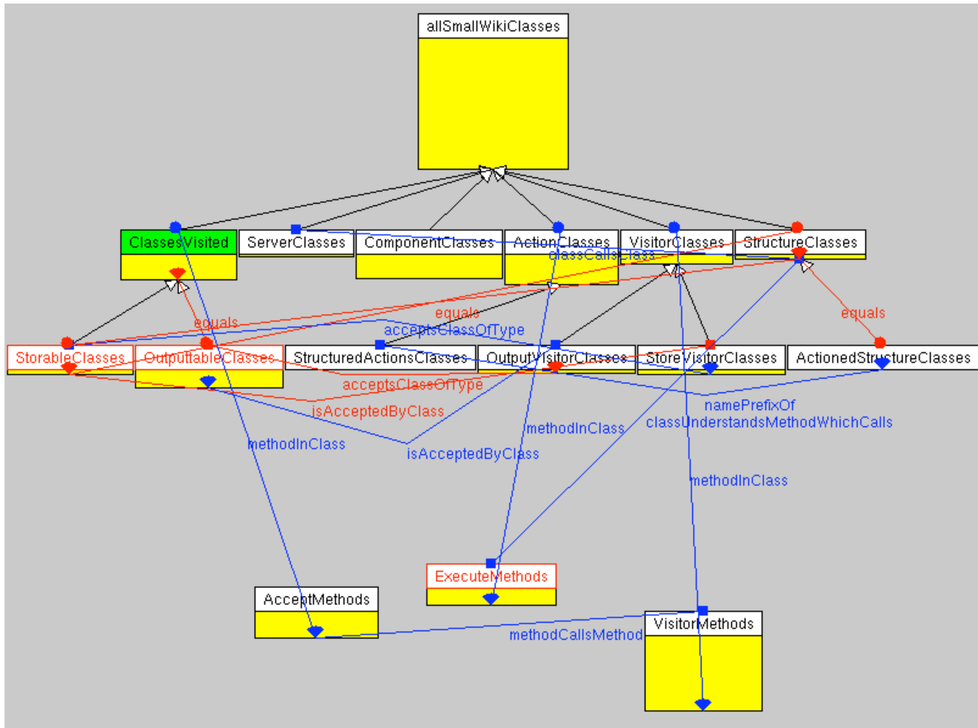

Fig. 5. The Intensional View Displayer at work on *SmallWiki* 1.304

sional views or relations. For example, a simple metric for a view is the number of entities contained in its extension. In Fig. 5 this metric was used as height of the rectangular boxes representing the views. For example, we can see that the view All SmallWiki Classes has many more entities than the Action Classes view, which is normal because the latter is defined as a subview of the former.

The visualization tool also uses colors to distinguish the different kinds of objects in a drawing. By default, the name and rectangle of intensional views are drawn in black, as well as the subview edges (starting with a triangle) and edges relaying a view with its alternative descriptions (ending with a diamond). The text and rectangle of the alternative descriptions are rendered in grey and an option can be toggled to not render them at all. Finally, edges representing intensional relations, together with the relation name, are drawn in blue. What is more interesting is that colors can be used as a metric too, for example to highlight inconsistencies in the documentation. The 'View Consistency' metric, for example, calculates the extensional consistency of a view and draws the view in red when inconsistent. A similar metric can be applied to the links connecting a view to its alternatives, to indicate what particular alternatives are inconsistent. In a similar way a color metric can be applied to the intensional relations, so that invalid intensional relations are highlighted in red. E.g., Fig. 5 immediately tells us that the view *Outputtable Classes* is inconsistent with some of its alternatives, and that the relation between *Classes Visited* and *Outputtable Classes* is invalid too: they are all colored red.

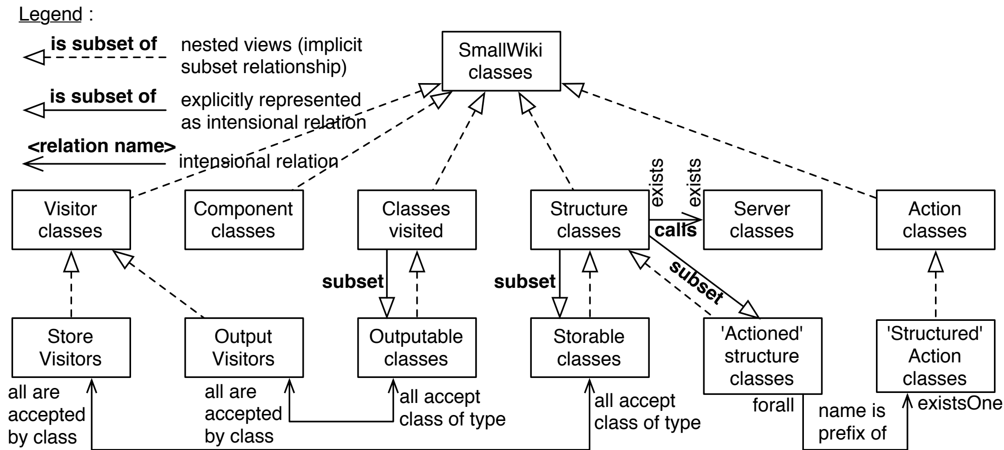

Fig. 6. Intensional Views and Relations on SmallWiki

4 Experiment 1 (Documenting the structure of *SmallWiki* 1.54)

Having explained the *IntensiVE* toolsuite in detail, we now elaborate on the actual experiments we conducted on *SmallWiki*. In our first experiment we tried to document the intended design of *SmallWiki* version 1.54 and investigated how this documentation helped us in better understanding the implementation structure as well as some of the naming and coding conventions that were used. Due to a lack of adequate documentation for this particular version, the approach we adopted was largely manual. We manually inspected the code, looking for interesting groups of classes or methods, codified those groups as intensional views, checked the views against the source code and further refined them when necessary, inspected the elements of the defined views to uncover relations with other views (potentially to be defined), etc.

In total, we came up with 17 intensional views, related by 14 nesting relationships and 16 intensional relations. Figures 6 and 7 summarize all defined views and relations. Whereas Fig. 6 shows the views containing classes and their interrelationships, Fig. 7 focusses on views containing methods. Next two subsections first discuss the views and then the relations between the views.

4.1 Views

All SmallWiki Classes. First of all we defined a view consisting of all classes in the application under study. This view was codified straightforwardly by means of a *Smalltalk* query `SmallWiki allClasses`.[3]

[3] Or alternatively using a *Soul* query `classInNamespace(?entity,[SmallWiki])`.

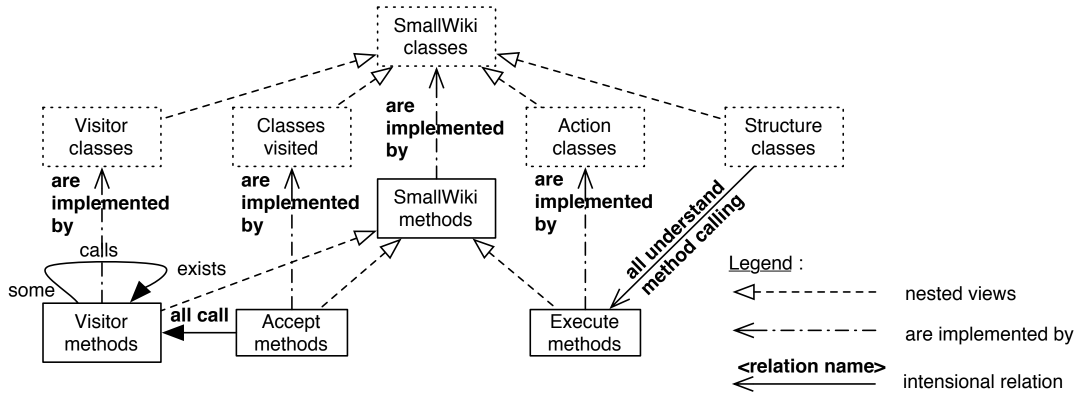

Fig. 7. Relations on SmallWiki method views

To restrict their domain to the *SmallWiki* classes only, the rest of the views were defined as subviews of this view. For example, we defined a series of views corresponding to the important class hierarchies in the code. They were all defined by means of a *Soul* query of the form `classInHierarchyOf(?entity,[`*root class of hierarchy*`])`.

Structure Classes (classes in hierarchy of `Structure`) represent *SmallWiki* entities that can be referred to by a single URL, like a web page.

Component Classes (`PageComponent` hierarchy) represent the components out of which a web page can be constructed: text, links, tables, lists, ...

Visitor Classes (`Visitor` hierarchy) visit the structure and component classes and play a crucial role in *SmallWiki*, e.g. for rendering and storing web pages.

Action Classes (`Action` hierarchy) model the actions that can be performed on Wiki pages.

Server Classes (`WikiServer` hierarchy) represent the different kind of Wiki servers supported by *SmallWiki* (version 1.54 only supported Swazoo).

Other views that we defined, mainly by manual code inspection, were:

Visitor methods are all methods implemented in the `Visitor` class hierarchy that belong to a visiting method protocol. In *Soul* this was expressed as:

```
classInHierarchyOf(?class,[SmallWiki.Visitor]),
methodOfClassInProtocol(?entity,?class,?protocol),
['visiting*' match:?protocol]
```

In fact this is an example of a hybrid query where we use logic to reason about the code structure and evaluate a *Smalltalk* expression, parameterized by a logic variable (this is a particular feature of *Soul*), to reason about strings.

Accept methods are the methods named accept: and play an important role in the Visitor design pattern. We defined this view by means of a *Soul* query methodWithName(?entity,[#accept:]).

Actioned structure classes We defined the group of all structure classes on which actions can be performed as a subview of the Structure Classes. They can be recognized easily because they have a corresponding Action class:

```
classInViewNamed(?c, ActionClasses),
['*Action' match: ?c name],
[(?entity name, 'Action') = ?c name asString]
```

Structured action classes Dually we defined the view of action classes for a particular structure class as a subview of the Action Classes.

Execute methods are responsible for executing different actions on Wiki pages, such as rendering, saving, canceling and editing. They have in common that their names start with 'execute'. We defined this view by means of the intension: ['execute*' match: ?entity selector asString]. Because we observed that the *SmallWiki* developer(s) consistently adopted the convention to put these methods in an 'action' method protocol, we also defined an alternative intension: methodInProtocol(?entity,action).

Defining the views above triggered the definition of some more views:

Store Visitors and Output Visitors After having defined the Classes Visited view we wondered what classes were being visited and for what reason. By inspecting the Visitor Classes in more detail we learned that in *SmallWiki* 1.54 there were two main visitors: a 'store' visitor and an 'output' visitor. We codified these straightforwardly as subviews of the Visitor Classes view: all Visitor classes named VisitorStore* or VisitorOutput*, respectively.

Storable Classes and Outputtable Classes We also defined a view representing the 'storable' classes, i.e. classes visited by a Store Visitor, and one representing the 'outputtable' classes, as subviews of Classes Visited. We only show the definition of the Storable Classes, the one for Outputtable Classes being analogous. We defined the view in terms of the newly defined Store Visitors: the store visitor classes need to implement a specific method accept<*name of class*>: for every class they want to visit. Without divulging all details, the following hybrid query extracts these visited classes from the names of the Store Visitors:

```
classInViewNamed(?class,StoreVisitors),
methodWithNameInClass(?method,?selector,?class),
[?selector = (#accept, ?entity name, ':') asSymbol]
```

As a second example of how existing views were reused to gradually refine and understand the code structure, the definition of the Visitor Classes view triggered the definition of a view consisting of all classes *being* visited:

Classes Visited are those classes that can be visited by Visitor Classes. Since the Visitor design pattern [10] uses a double dispatch protocol where the visited classes implement an accept method taking a visitor as argument, we defined this view using the *Soul* query `methodWithNameInClass(?M,[#accept:],?entity)`. In addition, since all these `accept:` methods belonged to a 'visiting' method protocol, as alternative description we used `protocolInClass(visiting,?entity)`.

When checking consistency of this view, we learned that the use of the 'visiting' protocol was indeed adhered to in a very disciplined way: all classes implementing `accept:` also had a 'visiting' method protocol and vice versa. To document this, we defined the following alternative for the previously discussed *Accept Methods* view: `methodInProtocol(?entity,visiting)`.

However, since all alternative descriptions should produce the same extension, this implied not only that every `accept:` method belongs to a `visiting` method protocol but also that every method in a `visiting` method protocol is an `accept:` method. That constraint was clearly too strong, as we learned when verifying it using the View Consistency Checker: the `Visitor` class did not implement an `accept:` method, but did contain a few 'visit' methods in the visiting protocol. By excluding the `Visitor` class from the new alternative, the constraint became valid.

We noticed that there are quite some views (and relations, as we will see in Subsection 4.2) in our design documentation that document the Visitor design pattern [10], even though that specific pattern is quite well known and well understood. Nevertheless, we decided to document it explicitly because of the crucial role the pattern plays in the *SmallWiki* implementation, but more importantly because we wanted to be able to verify whether the implementation constraints implied by this pattern remained consistently adhered to in future versions of *SmallWiki*.

4.2 Relations between intensional views

All relations we identified between intensional views containing classes are summarized in Fig. 6. Dashed lines ending with a triangle represent view nesting. In addition to those subset relationships we codified some extra subset relationships between non-nested views:

Classes Visited *is subset of* Outputtable Classes Not only are all outputtable classes a particular kind of visited classes (which was codified by means of nesting), in fact *all* visited classes are outputtable.

Structure Classes *is subset of* Storable Classes Whereas all visited classes are outputtable, only a few are storable. On the other hand, all structure classes, with the notable exception of the abstract superclass `Structure` itself, were storable. Since this seemed like a potentially important design contstraint, we documented it as an intensional relation with an explicit deviation for the exceptional case of the `Structure` class.

Structure Classes *is subset of* Actioned Structure Classes Although the 'actioned' structure classes were defined as subview of the structure classes, we observed that all structure classes (again with the exception of the class `Structure`) were 'actioned', i.e. had a corresponding `*Action` class.

'Actioned' Structure Classes versus 'Structured' Action Classes This same observation led us to define the following intensional relation between the 'Actioned' Structure Classes and 'Structured' Action Classes :

$\forall\ x \in \mathit{ActionedStructureClasses} : \exists!\ y \in \mathit{StructuredActionClasses} :$
$x\ has\ name\ which\ is\ prefix\ of\ name\ of\ y$

where the relation predicate was defined using the following *Smalltalk* block:

```
[:class1 :class2 | (class1 name asString),'*'
                    match: (class2 name asString)]
```

Next we defined the relationship between the visitors and the visited classes.

Output Visitors *all accept class of type* Outputtable Classes and
Store Visitors *all accept class of type* Storable Classes
Because of the double dispatch mechanism used in the visitor design pattern we know that *all* visitor classes that can handle a certain type of class need to implement *a* specific accept method taking objects of that type as argument. In particular this holds for the output visitors and outputtable classes, as well as for the store visitors and storable classes. The (hybrid) *Soul* predicate in terms of which we defined these intensional relations is given below. Due to the lack of static typing in *Smalltalk*, the predicate relies on the fact that the formal parameters of the method are named after the expected type.

```
acceptsClassOfType(?VisitorClass,?VisitedClass) if
   methodWithNameInClass(?Method,?Selector,?VisitorClass),
   ['accept*' match:?Selector asString],
   argumentOfMethod(?Argument,?Method),
   ['*',(?VisitedClass name asString),'*' match:?Argument asString]
```

Outputtable Classes *all are accepted by* Output Visitors and
Storable Classes *all are accepted by* Store Visitors
Conversely, all visited classes are supposed to be accepted by at least one visitor class. In particular this holds for the outputtable classes and output visitors, as well as for the storable classes and store visitors. The logic predicate in terms of which this relation is defined, is the inverse of the above:

```
isAcceptedByClass(?VisitedClass,?VisitorClass) if
   acceptsClassOfType(?VisitorClass,?VisitedClass)
```

We also documented that server classes are invoked by structure classes.

Structure Classes calls Server Classes Since not all server classes need to be invoked (it suffices to have one server running) and not all structure classes call the server classes, this intensional relation was defined as
$\exists\ x \in StructureClasses : \exists\ y \in ServerClasses : x\ classCallsClass\ y$
where x *classCallsClass* y checks if class x has a method that potentially calls a method on class y. This predicate was taken from the logic library.

Whereas Fig. 6 focused on views containing classes, Fig. 7 summarizes the relations between intensional views containing methods. First of all there are the obvious implementation relationships:

Accept Methods ***are Implemented By*** **Classes Visited**
Execute Methods ***are Implemented By*** **Action Classes**
Visitor Methods ***are Implemented By*** **Visitor Classes**

Other intensional relations which we documented were:

Accept Methods ***all call*** **Visitor Methods** Indeed, the `accept:` methods all have the following pattern to call an appropriate method on the visitor:

```
accept: aVisitor
   aVisitor accept<name of class>: self
```

To express this relation we used a universal quantifier and a predicate `method-CallsMethod` which we declared in *Smalltalk* using the following block:

```
[:method1 :method2 | method1 sendsSelector:
                     (method2 compiledMethod selector)]
```

Visitor Methods ***some call*** **Visitor Methods** This relation codifies the fact that a visitor method is often implemented in terms of other visitor methods. For example, quite some of the `accept*` visitor methods make a self call to the `visit:` method. For expressing this intensional relation we used the same predicate as above and a fuzzy quantifier *some* which requires that the relation is valid for at least 25% of the elements in its domain.

Structure Classes ***all understand method calling*** **Execute Methods**
Since actions are to be executed on things like web pages, which are represented by structure classes, we require that all these structure classes understand (at least one) method that calls an appropriate execute method for actually handling the actions. For example, the abstract class `Structure` implements a method named `evaluateActionWithRequest:response:` which calls `execute` on the appropriate action class. We defined this in terms of a logic predicate which we added to the logic library.

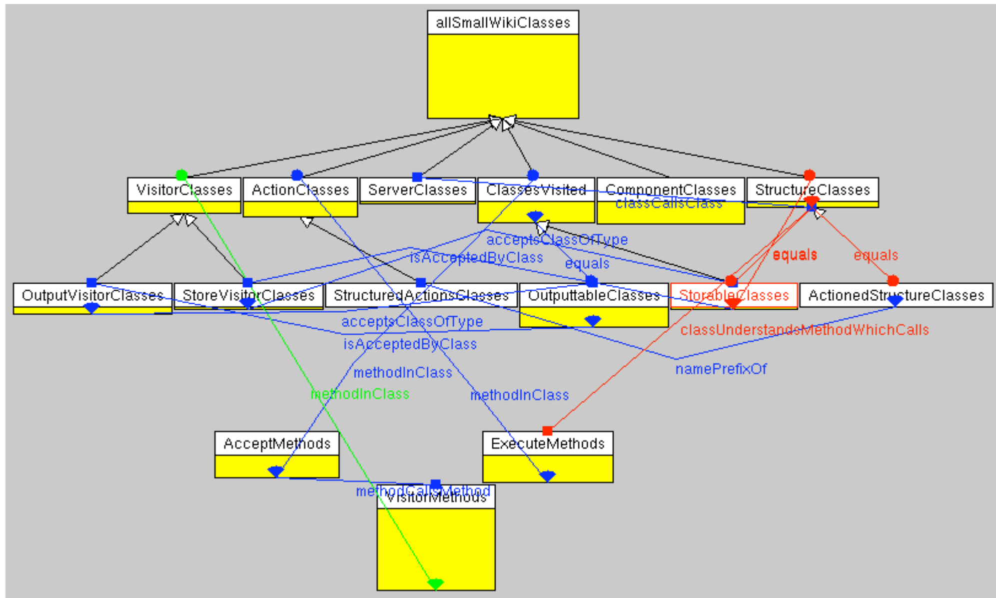

Fig. 8. Intensional views and relations on version 1.90

5 Experiment 2 (Comparing the documentation with *SmallWiki* 1.90)

In the second experiment we compared the documented design of version 1.54 to the more recent version 1.90 and tried to understand how *SmallWiki* evolved, and what the consequences of this evolution were on the design documentation. To do so, we loaded the new version and recomputed and visualized all known intensional views and relations with the Intensional View Displayer. As explained in Subsection 3.5 and illustrated in Fig. 8, all conflicting views and relations were highlighted in red. We inspected the conflicts and tried to understand the discovered problems.

Inconsistent views

Storable Classes became inconsistent because of an explicit deviation in the third alternative that was no longer needed. More precisely, we originally documented that all Storable Classes except the `Document` class were Structure Classes. In version 1.90, the system had been refactored such that the `Document` class was moved to another hierarchy. We updated the design documentation by removing the deviation.

Invalid relations

Structure Classes *all understand method calling* Execute Methods failed because in *SmallWiki* 1.54 the execute methods were being abused to render web pages in HTML format. In the newer and cleaner version, page rendering was performed by separate rendering methods. The only remaining purpose of the `execute` methods was to dispatch action requests to more appropriate methods

depending on the action to be taken. Hence the invalid relation highlighted an interesting restructuring of the application. To update the documentation we did two things:

(1) we documented the dispatching mechanism by defining an intensional relation which required the `execute` methods to make a self call to a more specific `execute*` method (where `*` is a non-empty string). E.g., the method `execute` on class `ChapterEdit` calls either `executeCancel` or `executeSave` if the corresponding button was selected and performs an `executeEdit` otherwise;
(2) we documented that the `execute` methods were not allowed to send messages to the instance variable named `html` (which was typically the way how rendering was being done).

After having done so we still found a few violations against these new constraints but did not codify them as explicit deviations, since we wanted to emphasize that they were real design conflicts that should be fixed in a new version of the code.

Structure Classes *is subset of* Storable Classes failed because of the addition of two new intermediate superclasses. We defined these two classes as deviating cases of the relation.

Comparing view sizes

Using the Intensional View Displayer, we compared the size of all views on version 1.54 with those on version 1.90. We wanted to find out if and where there were important differences in size, as these may indicate potential problems. We did this by using the Intensional View Displayer and choosing the number of entities in the extension as height of the view. There didn't appear to be any real problems except for the Actioned Structure Classes view and its dual view the Structured Action Classes which both became empty. When trying to understand the reason we found out that the view definition needed refinement. The introduction of some intermediate classes in version 1.90 forced us to use the `classInHierarchyOf` predicate instead of `subclassOf`.

Newly introduced views and relations

Because of the restructuring of the code in version 1.90, we needed to add one new view and one new relation:

Rendering Methods. The restructuring of the Execute Methods made us decide to define a new intensional view grouping all **Rendering Methods**.

Execute Methods *call* Rendering Methods The restructuring caused the responsibility of rendering web pages to be shifted from the execute methods to the rendering methods, but rendering was still triggered by the execute methods.

6 Experiment 3 (Verifying the design structure of *SmallWiki* 1.304)

In the third and last experiment we reverified our design documentation on yet a more recent version of SmallWiki (the one visually represented in Fig. 5) and drew conclusions about the usefulness of intensional views and relations to document the design structure of an evolving software system over a longer development period. Again, the design documentation appeared to be quite stable, but nevertheless we discovered some interesting inconsistent views and invalid relations, which are discussed next. We also compared the view sizes with those on the previous version.

Inconsistent Views

Outputtable Classes became inconsistent because of the addition of four new classes. Instead of having a specialized `accept*` method like the other `Outputtable Classes` (and as expected by the view definition), these classes delegated their accept method to a more general one. We solved this by refining the view such that it is declared as the conjunction of the classes with a specialized accept method together with the classes that delegate their accept method.

Storable Classes became inconsistent, as can be seen from Fig. 5, for the same reasons as the `Outputtable classes` view. By redefining this view in an analogous way, the consistency of this view was restored.

Execute Methods is no longer consistent because some coding conventions were not adhered to consistently in this version: there were two 'execute' methods that were not implemented in the correct protocol, and there were two other methods that were in the right protocol but did not start with the string 'execute'. To fix the problem the former just needed to be moved to the correct protocol whereas the latter either needed to be renamed or put in a more appropriate protocol.

Invalid Relations

Structure Classes *is subset of* Storable Classes and
Classes Visited *is subset of* Outputtable Classes failed because of the failure of the Storable Classes and Outputtable classes views, as discussed above. After fixing these views, these relations became valid again.

Storable Classes *are all accepted by* Store Visitors failed since the argument of the accept method on `LinkInternalVisitor` was called '`anInternalLink`' instead of on the expected '`aLinkInternal`'. (Remember that the predicate definition relied on the fact that the argument names respected a particular naming convention.) We fixed this problem by renaming the argument.

Store Visitors *all accept class of type* Storable Classes failed due to the addition of new storable classes which were not taken into account by the `Storable Classes` view. We solved this conflict by extending the `Storable Classes` view.

Output Visitors ***all accept class of type*** **Outputtable Classes** failed because in the original version of this intensional relation we documented the classes `AnObsoleteVisitorOutput` and `AnObsoleteVisitorHtml` as explicit deviations of the relation. These classes however were removed from the code between experiment 2 and 3 and thus caused this relation to fail. This was fixed by removing the deviating cases from the documentation again.

Comparing view sizes

We compared the sizes of the (extension of the) intensional views on version 1.90 with those on version 1.304 and observed two important differences: the number of Action Classes almost doubled (from 13 to 25), because more functionality had been added to *SmallWiki*, whereas the number of Execute Methods further diminished from 23 to 14, illustrating the continued migration from the old style of execute methods to those using the visitor pattern.

7 Critical analysis and lessons learned

In this section, based on our experiences gained with the *SmallWiki* case, we perform a critical analysis of the current generation of tools — including the new opportunities offered by the visualization tool — and of the underlying model of intensional views and relations, to support co-evolution of high-level design and source code of a medium-sized *Smalltalk* application.

Deviations The experiments illustrated the importance of being able to define explicit deviations (inclusions and exclusions) to intensional views and relations. This happened when the implementation should have adhered to an intension or relation, but for various reasons did not. Typically, this either indicated an opportunity to refactor the code, or to refine an intension that was expressed too broadly. In either case it was useful to document the deviating cases explicitly. When eventually fixing the code or intension and reverifying consistency, the tool would issue warnings about deviations that had become obsolete, confirming us that the exceptional case had indeed been solved, at which point we could safely remove the corresponding deviation.

Completeness Although intensional views and relations allowed us to express and verify interesting structural constraints about the source code, the obtained design documentation was by no means complete. For example, it could prove useful to complement this design documentation with more dynamic information produced by other tools.

Static versus dynamic information Indeed, both query languages supported by our tool (*Soul* and *Smalltalk*) allowed us to define views and relations which reason about the *static structure of a system only.* Although we did not experience the lack of dynamic information as a severe hindrance while documenting the

design of *SmallWiki*, we do agree that this restriction may prohibit us in documenting some interesting design constraints. For instance, the concept of a layered architecture is very hard to express without the use of dynamic information.

Incremental approach In our experiment, we adopted an incremental approach to document *SmallWiki*. Starting from a minimal working knowledge about the case, we gradually refined and documented our knowledge about the system by alternating manual code inspection with the definition of views and relations and verifying them against the source code. The tools helped us in codifying and testing our assumptions about the code structure and in finding out where the assumptions were (or became) invalid and why. This incremental approach not only allowed us to obtain a fine-grained documentation of the structure of *SmallWiki*, but at the same time helped us in obtaining a better comprehension of the system's implementation. In addition, we observed that verifying the documentation against newer versions of the code often provided us with valuable insights in how the application's design evolved.

Co-evolution The goal of the *IntensiVE* toolsuite is to support co-evolution of code and design documentation. To this end, our tools support the detection of structural conflicts between documentation and code, when either of them have evolved. We can discriminate between two kinds of conflicts. A first kind of conflict is when the documentation is conceptually correct, but some parts in the code violate it. This can happen when an actual bug was introduced in the code (e.g., removing a method that is being relied on) or when a certain naming or coding convention (e.g., putting a method in the wrong protocol) or architectural constraint was violated (e.g., adding a class that can be visited but does not implement the appropriate methods). In order to fix these conflicts, the code needs to be adapted. The other kind of conflicts that may occur are caused by code restructurings that affect the original design documentation. Such conflicts typically need to be solved by modifying the design documentation, i.e. adapting the views and relations.

Perhaps surprisingly, the majority of conflicts we detected were of the second kind, i.e. they were caused by code restructurings of *SmallWiki*. Indeed, over the different versions of *SmallWiki*, the source code was often restructured in order to improve the design of the application. A possible explanation for the fact that we did not discover many conflicts of the first kind is that we did not apply the documentation to a system under development, but rather applied it 'a posteriori' to versions of *SmallWiki* which had already been released and tested.

Visualization One of the most recent additions to *IntensiVE* is the visualization tool. By making good use of the underlying *CodeCrawler* tool, we could use it not only to display the declared views and relations, but also to highlight inconsistent views and relations and to help us assess the impact of an evolution of the system. Using the *CodeCrawler* integration, with an appropriate metric we could for instance visualize the size of the views and the cardinality of the differences between the various alternatives of a view. This was a significant improvement over earlier versions of our tools where we had to manually inspect all views and relations in order to get an idea of the impact of evolution on the documentation.

A downside of using the visualization was that, when the number of views and relations increased, the visual representation became cluttered. A pragmatical solution to this problem was to visualize only a selection of views and relations.

Choice of query language An interesting question when using *IntensiVE* is what query language to select. When defining an intensional view or relation, should we prefer logic queries over *Smalltalk* queries, or perhaps prefer hybrid queries? The rule of thumb we adopted was to always choose the language that best suited our needs, that is, the language in which we could express the query or predicate in the most compact, yet still declarative way. In practice, it often turned out that a hybrid query was most appropriate. For example, we could have defined the Structured Action Classes view by means of a logic query:

```
classWithName(?entity,?ename),
endsWith(?ename,['Action']),
classInViewNamed(?c,StructureClasses),
classWithName(?c,?cname),
equals([?cname, 'Action'], [?ename asString])
```

By using a mixture of logic and *Smalltalk* code, however, we could write the query much more compactly, by doing the string pattern matching in *Smalltalk* and the reasoning about the code structure in logic:

```
['*Action' match: ?entity name],
subclassOf(?c, [SmallWiki.Structure]),
[(?c name, 'Action') = ?entity name asString]
```

In an extreme case this even resulted in a hybrid query which took 4 lines of code, while the same query, written down in *Smalltalk* took 17 lines.

Nevertheless, without going in the technical details, when using *IntensiVE* we did occasionally notice some limitations when trying to mix queries and predicates defined in the different languages. To solve these limitations, a better integration and symbiosis of the logic and *Smalltalk* query languages and libraries is required (like the one proposed in [11]).

Is logic programming needed? On the other hand, none of the declared views or relations in this case study required the full power offered by our logic programming language *Soul*. Hence we could probably use a less expressive but faster query mechanism like SmallLint [12], and still be able to codify the same views. But then we would also loose the abstraction facilities offered by our logic programming language, as well as its logic library containing an extensive set of predicates to reason about Smalltalk source code.

8 Conclusion

This paper investigated how the model of intensional views and relations and the *IntensiVE* toolsuite can be used to support co-evolution of source code and design of

a software system. The evaluation was done by documenting the design of an early version of *SmallWiki* and checking this documentation against two more recent versions of *SmallWiki*. Doing these experiments we observed that:

- Although building a first version of the design documentation of an unknown system remains a largely manual process, the incremental nature of the approach, combined with tool support to verify and visualize conformance of the design against the code, helps us in understanding the code and its structure.
- Once the design of a system has been documented with intensional views and relations, conformance of this design against other versions can be checked and visualized. Even by simply reverifying the defined views and relations on another version of the software, we gain useful insights on how the software evolved.
- Visualization of high-level design documentation is useful and important, especially when combined with advanced metrics and coloring to highlight potential inconsistencies. In a glimpse of the eye it is possible to get an overview of the design, and assess whether it conforms to the code, and where not.
- Being able to use different query languages to express views and relations is important. It means that the language most appropriate to express certain kinds of information can be chosen. At the same time it reduces the learning cost of the approach: someone not proficient with logic programming can start with simple *Smalltalk* queries and gradually learn to use the logic language and library. A good integration and symbiosis of the query languages and libraries is essential, however.

Overall, despite some minor limitations of the environment, the *IntensiVE* toolsuite supported us quite well in documenting the high-level structure of *SmallWiki* and keeping it synchronized with the code as it evolved, while at the same time providing us with useful insights on how the code structure evolved over time.

References

[1] A. J. Ko, H. H. Aung, B. A. Myers, Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance, in: Proceedings of the International Conference on Software Engineering ICSE'2005, IEEE Computer Society, 2005, pp. 126–135.

[2] K. Mens, R. Wuyts, T. D'Hondt, Declaratively codifying software architectures using virtual software classifications, in: Proceedings of TOOLS Europe 1999, IEEE Computer Society Press, 1999, pp. 33–45, TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

[3] K. Mens, T. Mens, M. Wermelinger, Maintaining software through intentional source-code views, in: Proceedings of the International Conference on Software

Engineering and Knowledge Engineering (SEKE'02), ACM Press, 2002, pp. 289–296.

[4] K. Mens, B. Poll, S. González, Using intentional source-code views to aid software maintenance, in: Proceedings of the International Conference on Software Maintenance (ICSM'03), IEEE Computer Society Press, 2003, pp. 169–178.

[5] K. Mens, A. Kellens, Towards a framework for testing structural source code regularities, submitted to ISCM 2005.

[6] R. Wuyts, S. Ducasse, Unanticipated integration of development tools using the classification model, Computer Languages, Systems and Structures 30 (1-2).

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

[8] M. Lanza, Codecrawler: Lessons learned in building a software visualization tool, in: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), IEEE Computer Society, 2003, pp. 409–418.

[9] L. Renggli, Collaborative web : Under the cover, Master's thesis, University of Berne (2005).

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Languages and Systems, Addisson-Wesley, 1994.

[11] K. Gybels, Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis, in: Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.

[12] D. Roberts, J. Brant, R. Johnson, B. Opdyke, An Automated Refactoring Tool, in: Proceedings of ICAST 1996, Chicago, IL, 1996.