

# Towards a Framework for Testing Structural Source-Code Regularities

Kim Mens

Département d'Ingénierie Informatique  
Université catholique de Louvain  
Place Sainte Barbe, 2  
B-1348 Louvain-la-Neuve, Belgium  
kim.mens@info.ucl.ac.be

Andy Kellens\*

Programming Technology Lab  
Vrije Universiteit Brussel  
Pleinlaan 2  
B-1050 Brussels, Belgium  
akellens@vub.ac.be

## Abstract

*As size and complexity of software systems increase, preserving the design and specification of their implementation structure gains importance in order to maintain the evolvability of the system. However, due to constant changes, the implementation structure and its documentation tend to dilute over time. Building on the underlying models of intensional views and intensional relations, our IntensiVE tool-suite helps a developer in documenting structural source-code regularities, verifying them and offering fine-grained feedback when the code does not satisfy those regularities.*

**Keywords:** structural source-code regularities, intensional views and relations, design documentation, automated conformance checking, maintenance and evolution, tool support.

## 1 Introduction

Due to changing requirements, bug-fixes or adoption of new technology, software systems constantly evolve. The ever increasing size and complexity of software however renders the task of evolving a software system a non-trivial one, making it imperative for the design documentation and implementation structure of the system to be up to date and explicitly known to developers and maintainers. Unfortunately, the quality of the structure and documentation of the system tend to decay over time, thus having a negative impact on the overall maintainability of the system. To alleviate this problem, we developed the *IntensiVE* tool-suite, based on the underlying model of intensional views [6] and relations. It allows for the documentation of structural source-code regularities, like nam-

ing conventions, programming conventions and structural dependencies, that are shared by multiple source-code entities (classes, methods, packages) spread throughout the program. More importantly, the tool-suite offers support to verify conformance of that documentation to the implementation and to provide fine-grained feedback when inconsistencies between documentation and implementation are discovered.

## 2 Intensional Views

We explain the model of intensional views using the Visitor design pattern [2]. The Visitor is an object-oriented design pattern that can be applied to implement a variety of different operations on a hierarchy of elements, while keeping the operations' implementation independent of the element hierarchy. The pattern is implemented by providing, on all classes of the element hierarchy, an `accept` method which takes as argument an instance of a visitor class (representing the operation to be carried out on the elements) and which calls a corresponding `visit` method defined on the visitor class, using a so-called "double-dispatch" protocol. [2]

An *intensional source-code view*, or intensional view for short, is a set of source-code entities (e.g., classes, methods) that share an arbitrary, but well-defined structural property. Instead of defining such a set by explicitly enumerating all of its elements, it is defined by specifying an *intension*: an executable description which codifies the commonalities of all entities belonging to the view. The intension of a view is described in *Soul* [5], a dedicated logic programming language that can query and reason about object-oriented (Smalltalk) source code. Evaluating a view's intension produces its *extension*: the set of entities that satisfy the description. Fig. 1 shows the extension of two simple intensional views on the Visitor example: the *Accept*

\*Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

*Methods* view which groups all `accept` methods and the *Visit Methods* view which groups all methods whose name start with ‘visit’ and which are implemented on a subclass of `AbstractVisitor`. (The alternative *Accept Methods* view and barred method are explained below.)

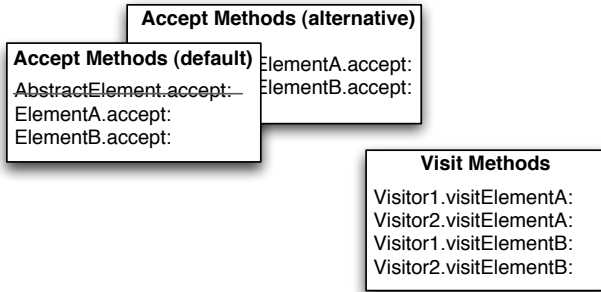


Figure 1. Two views on the Visitor

The model of intensional views also supports the definition of multiple, *alternative* intensions for the same view, one of which is called the *default* alternative. We can think of two different intensions for the *Accept Methods* view:

1. All methods named ‘accept’ taking a single parameter;
2. All methods implemented by a subclass of `AbstractElement` which perform a double dispatch.

When specifying multiple alternatives for a view we require them to be *extensionally consistent*: upon evaluation of its intension, each alternative should yield the same extension.

Fig. 1 shows the extensions of both alternatives of the *Accept Methods* view. Notice that the two alternatives above are *not* extensionally consistent. Alternative 1 is more general: it includes an abstract accept method which does not satisfy alternative 2, since an abstract method has no implementation. To deal with conflicting cases like these, our model supports the annotation of each alternative with an *inclusion* and *exclusion* set which allow users to indicate explicitly what entities need to be included, respectively excluded, from the extension produced by that alternative. For example, we make the two alternatives above extensionally consistent, by excluding the conflicting abstract method from alternative 1. We depicted this in Fig. 1 by barring that method. The requirement of extensional consistency allows us to express some interesting structural source-code regularities, and the inclusion and exclusion sets allow us to document explicitly what source-code entities do not satisfy such regularities.

### 3 Intensional Relations

*Intensional relations* are binary relations between intensional views of the following canonical form:

$$Q_1 x \in V_1 : Q_2 y \in V_2 : x R y$$

where  $Q_1$  and  $Q_2$  are logic quantifiers  $\forall, \exists, \exists!$  or  $\nexists$ ;  $V_1$  and  $V_2$  are intensional views and  $R$  is a binary relation over the source-code entities (denoted by  $x$  and  $y$ ) contained in those views. For example, in the Visitor pattern an important intensional relation holds between the *Accept Methods* view and the *Visit Methods*: every accept method calls a corresponding visit method. Formally, we have:

$$\forall x \in \text{Accept Methods} : \exists! y \in \text{Visit Methods} : x \text{ methodDoesSend } y \quad (1)$$

where  $x \text{ methodDoesSend } y$  is a binary relation over source-code entities that holds when  $x$  and  $y$  are methods and  $x$  sends a message to  $y$ .

What actual source-code relations  $R$  are supported and how they are implemented depends on the chosen query language. In our logic meta-programming language *Soul* [5], we can use as relation  $R$  any binary predicate provided by LiCoR, a library of predicates for reasoning about static source-code dependencies like method implementations, message sending, and so on.

### 4 Using *IntensiVE* to codify and test structural source-code regularities

Using a concrete instantiation of the Visitor design pattern in a Smalltalk program as an example, we now explain how our *IntensiVE* toolsuite supports a software engineer in documenting structural regularities and co-evolving this documentation with the source code. For more information on *IntensiVE* we refer to [3]. For a larger case study, we refer to [4].

#### Codifying structural regularities

When coding, a software developer often takes important decisions about the program structure. When trying to understand a program a software engineer makes a mental picture of the program’s structure. In either case, there is a need to store this knowledge explicitly, so that the knowledge does not get lost, so that it can be communicated to others, and so that it can be checked whether the code conforms to that knowledge, or found out where it does not.

The *IntensiVE* toolsuite enables a software developer to document explicitly, and incrementally, the structural

regularities in a program. Whenever a developer discovers a structurally relevant group of source-code entities or a structural relationship between such groups, he can try to codify it as an intensional view or intensional relation, respectively. E.g., in the Visitor pattern, knowing that all *Accept Methods* are structurally similar, we group all these methods in an intensional view, as explained in Section 2.

The following *SOUL* query describes the default intension of the *Accept Methods* view:

```
classInHierarchyOf(?c,[AbstractTerm]),
methodWithNameInClass(?entity,[#accept:],?c)
```

This logic query declares that a source-code *?entity* is part of the view if it is a method named *accept:* implemented on a class in the *AbstractTerm* class hierarchy (the element hierarchy of the Visitor pattern).

Having defined this intensional view, we inspect it in more detail by exploring its extension, and discover that, with the notable exception of the abstract *accept:* method defined on *AbstractTerm*, all methods in this view have exactly the same format:

```
accept: aVisitor
  ^aVisit visit<name>: self
```

where *<name>* is the name of the class implementing the method. Since this “double dispatch” protocol is an important coding convention we decide to encode it as follows:

1. We explicitly exclude the *accept:* method on *AbstractTerm* from the default intension, as it has no concrete implementation.
2. We define an alternative intension specifying that all methods in the view have the above format.
3. We define a new intensional view *Visit Methods* that groups all the visit methods.
4. We explicitly codify the intensional relation (1) which states that every *accept* method calls a visit method.

### Check conformance

Whenever we have documented some structural source-code regularities, we can use our toolsuite to check whether the source code actually conforms to those regularities. The two main mechanisms for doing so are: (a) defining multiple alternative intensions of a view and verifying extensional consistency among those alternatives, and (b) defining and verifying intensional relations between views.

If the conformance check succeeds, we know that we have correctly documented a structural regularity. If the check fails, more fine-grained information about what went wrong will be produced, as we will see in Subsection 4. In that case, there are basically three different ways in which a software developer can solve the problem:

1. When the codified regularities were not entirely correct or not sufficiently precise, he can refine the intensional views and relations that document these structural regularities, using the same tools as before ;
2. When the codified regularities were conceptually correct but the source code does not consistently satisfy these regularities, the developer may wish to restructure the source code so that it does. After having modified the code, he can recheck the regularity.
3. When the developer lacks sufficient knowledge to modify the code immediately, he can explicitly annotate the inconsistencies as ‘known deviations’.

We experienced that in practice strategy 3 is often useful as a temporary fix when we get in trouble with strategy 1 or 2. Of course we should return to those strategies later to get to the heart of the problem and remove the documented deviations.

### Co-evolution of source code and structure

In order for a software developer to modify the source code or the declared structural regularities in such a way that both become or remain consistent, it is imperative that he receives fine-grained feedback on where the source code violates the regularities. The *IntensiVE* toolsuite contains two dedicated tools that provide such fine-grained feedback.

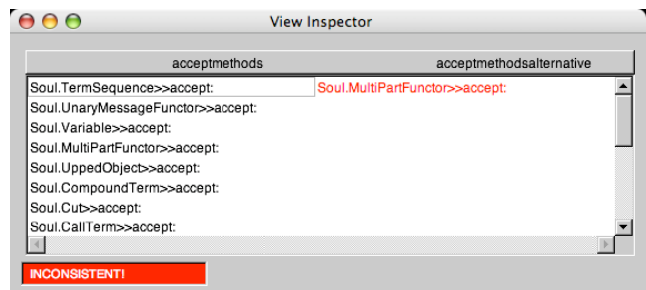


Figure 2. The View Inspector

The **View Inspector** (Fig. 2) is launched whenever checking extensional consistency of a view fails. The first column lists all source-code entities that satisfy the default intension of the view. The other columns<sup>1</sup> show the delta between the default intension and each of the alternative intensions. When applied to the *Accept Methods* view, we discovered a method named *accept:* on class *MultiPartFuncor* which satisfied the default intension but not the alternative one. We discuss later how we solved this inconsistency.

<sup>1</sup>In Fig. 2 there are only 2 columns but in general there are as many columns as there are alternative views.

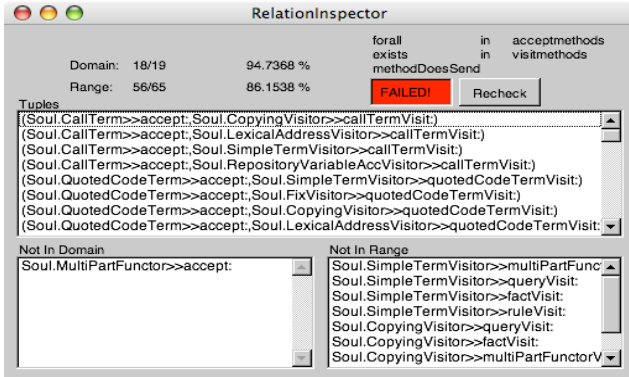


Figure 3. The Relation Inspector

The **Relation Inspector** (Fig. 3) is launched whenever an intensional relation is checked. In addition to indicating whether the relation succeeded, it shows all tuples of source-code entities for which the relation predicate, in terms of which the relation is defined, holds. Those entities in either the source or target view which do not appear in any tuple are indicated in the bottom two panes. The amount and percentage of entities in both the source and target view that participate in the relation are also shown.

When applying the Relation Editor on the ‘call’ relation (1) between *Accept Methods* and *View Methods* (Fig. 3), we learned that the relation failed because the method `accept:`, implemented by the `MultiPartFuncor` did not call a visit method. This is the same method that caused the extensional consistency of the *Accept Methods* to fail. When inspecting the code of that method we saw that it is in fact an abstract method. We explicitly exclude it from the default alternative of the *Accept Methods* view, just like we did with the abstract `accept:` method on `AbstractTerm`, which resolved both inconsistencies.

### Methodological aspects

Although we explained each of the above activities separately, we do not regard them as separate activities that should be performed sequentially. Rather, we see them as part of an incremental and iterative process where documentation, conformance checking and co-evolution of the structural regularities are strongly intertwined.

We purposefully designed *IntensiVE* as a non-coercive set of tools in a software developer’s toolbox. It is the developer who decides whether, when and how to use them. Based on our experience with *IntensiVE* we advocate an incremental and iterative methodology which bears some resemblance with XP testing [1]:

- Intensional views and relations *document* important structural constraints and dependencies in the source

code. The developer documents, checks and refines these structural regularities *by need*, whenever he feels there is a need to do so.

- The documented regularities are relatively *isolated*: every intensional view can be checked independently for extensional consistency, and every intensional relation can be verified independently of any other. Modifying a view, however, may invalidate some of the intensional relations in which it participates directly.
- Documenting and checking the structural regularities helps us in better *understanding* the source-code structure and at a same time give us *confidence* that the software is structured as desired.
- Even though it may require some insight to correctly define an intensional view or relation, our *IntensiVE* toolsuite has been designed as a *lightweight* set of tools that are *seamlessly integrated* with the development environment and that incite developers to document structural regularities and *check them frequently*.

## 5 Summary

In this paper we demonstrated a methodology similar to XP testing for documenting structural regularities in source code, supported by our *IntensiVE* toolsuite. To illustrate the approach, we documented an instance of the Visitor design pattern and demonstrated how our tools help in co-evolving the documentation and implementation of the pattern.

## References

- [1] K. Beck. *Extreme programming eXplained : embrace change*. Addison-Wesley, 2000.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison-Wesley, 1994.
- [3] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. The intensional view environment. In *Proceedings of Industrial Application and Tool Demonstration papers of the International Conference on Software Maintenance ICSM 2005*, 2005.
- [4] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views - a case study. *Elsevier Journal on Computer Languages, Systems & Structures*, 2006 (to appear).
- [5] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, November 2002.
- [6] K. Mens, B. Poll, and S. González. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM’03)*, pages 169–178. IEEE Computer Society Press, 2003.