IntensiVE, a toolsuite for documenting and checking 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. To address this problem, we developed IntensiVE: a toolsuite for documenting and checking structural source-code regularities. Building on the underlying models of intensional views and relations, the toolsuite helps a developer in documenting structural source-code regularities, verifying them and offering fine-grained feedback when the source-code does not satisfy those regularities. By illustrating our tools on a Smalltalk application, we show that violations of the source code against the structural regularities can be detected easily and that our toolsuite provides useful feedback for a developer to refine the regularities or to fix the code so that it does satisfy the regularities.

1 Introduction

Due to changing requirements, bugfixes or the adoption of new technology, software systems constantly evolve. The ever increasing size and complexity of software 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 decrease over time, thus having a negative impact on the overall maintainability of the system. To alleviate this problem, we developed the *IntensiVE* toolsuite based on the underlying model of intensional views [11] extended with a model of intensional relations. It allows for the documentation of structural source-code regularities like naming conventions, programming conventions and structural dependencies, that are shared by multiple source-code entities (classes, methods, packages) spread throughout a program. More importantly, the toolsuite 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. This paper provides the following contributions:

- The definition of *intensional relations*, an extension to the model of *intensional views*;
- An overview of some tools of the *IntensiVE* toolsuite;
- The description of a step-wise methodology, bearing resemblance with eXtreme Programming testing, on the usage of our toolsuite to support documentation and conformance checking of structural source-code regularities;
- An illustration of how the feedback provided by *IntensiVE* can be used to diminish the drift between design documentation and implementation.

The remainder of this paper is structured as follows. In Section 2 we repeat the model of *intensional views* and extend it with *intensional relations*. Our *IntensiVE* toolsuite for documenting and checking structural source-code regularities is explained in Section 3. Section 4 validates the tools by documenting and checking the structure of the *DelfSTof* application. Section 5 discusses the results of this case study and proposes avenues for future research. We conclude the paper after having presented related work in Section 6.

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

2 Intensional Views and Relations

We introduce the models of *intensional views* and *intensional relations*, which underly our *IntensiVE* toolsuite, using the running example of the Visitor design pattern [5].

2.1 Intensional Views

The Visitor (Fig. 1) is an object-oriented design pattern that is used 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. [5]

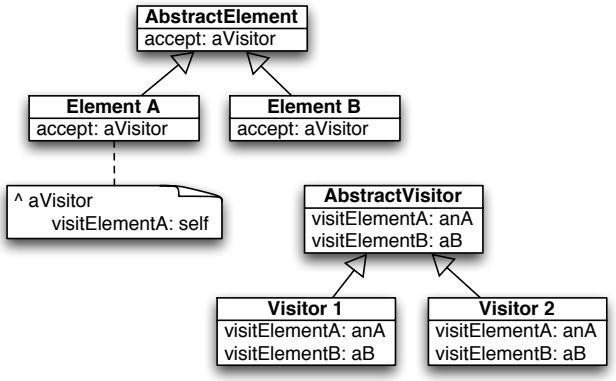

Figure 1. The Visitor Design Pattern

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. Evaluating a view's intension produces its *extension*: the set of entities that currently satisfy the description. Fig. 2 shows the extension of two simple intensional views on the Visitor example: the ***Accept 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 intension of a view is described either in Smalltalk, or in *Soul* [10], a dedicated logic programming language that can query and reason about object-oriented (Smalltalk)

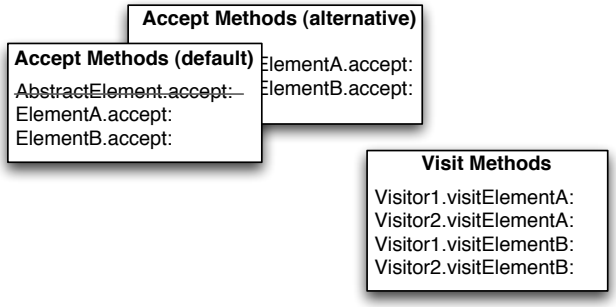

Figure 2. Two views on the Visitor

code. 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 on 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. Verifying extensional consistency thus straightforwardly amounts to computing the solution sets of the queries corresponding to each of the alternatives and then checking equality of these sets.

Notice that the two alternatives above are *not* extensionally consistent. Fig. 2 shows the extensions of both alternatives of the ***Accept Methods*** view. Alternative 1 is more general: it includes an abstract accept method which does not satisfy alternative 2, since the abstract method has no implementation and thus not perform the double dispatch. 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 declare explicitly what deviating entities need to be included, respectively excluded, from the extension produced by that alternative. For example, to make the two alternatives above extensionally consistent, we exclude the conflicting abstract method from alternative 1. We depicted this in Fig. 2 by barring that method. We will see later how the requirement of extensional consistency allows us to express some interesting structural source-code regularities, and how the inclusion and exclusion sets allow us to document explicitly what source-code entities do not satisfy such a regularity.

2.2 Intensional Relations

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

$$\mathcal{Q}_1\ x \in V_1 : \mathcal{Q}_2\ y \in V_2 : x\ R\ y$$

where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ 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:

$$\begin{array}{l}\forall x \in \texttt{Accept Methods}: \\ \quad \exists! y \in \texttt{Visit Methods}: \\ \qquad x\ methodDoesSend\ y\end{array} \quad (1)$$

where $x\ methodDoesSend\ y$ is a binary relation over source-code entities that holds when x and y are methods and x sends a message to y. Verifying such an intensional relation against the source code amounts to evaluating the corresponding expression, where the intensional views have been substituted by their extension.

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*, we can use as relation R any binary predicate provided by *Soul* [10]. For practical purposes, we restricted ourselves to 15 predicates which reify static source-code dependencies like method implementation and overriding, message sending, return statements, inheritance and class referencing.

3 Using IntensiVE to document and check structural source-code regularities

Using a concrete instantiation of the Visitor design pattern in as an example, we now explain the *IntensiVE* toolsuite[1], implemented in Visualworks Smalltalk, and how it supports carrying out three main activities:

1. Documenting the structural regularities in a program by a developer or encoding of initial hypotheses about the program structure during software comprehension.
2. Checking conformance of the source code to the intended structure or documented initial hypotheses.
3. Co-evolution of source code and structural regularities when either of them evolve, by providing fine-grained feedback on what parts of the source code invalidate the structural regularities.

[1]The most recent version of our IntensiVE toolsuite is available for download from *www.intensional.be*.

Although we explain each of these activities separately below, 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.

3.1 Documenting 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 explicitly and incrementally document 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.

Fig. 3 shows the **Intensional View Editor** tool opened on the default intension of the ***Accept Methods*** view. This tool supports the definition, evaluation, inspection and conformance checking of intensional views. The selected query language is the logic meta-programming language *Soul*. 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 evaluating and 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 encode it as follows:

1. We explicitly exclude the `accept:` method on `AbstractTerm` from the default intension, as it has no concrete implementation. See Excludes pane in Fig. 3.

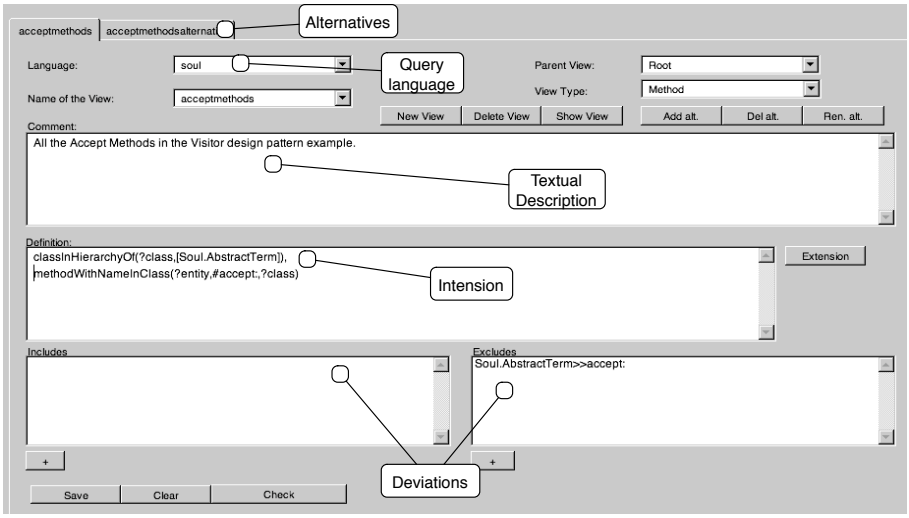

Figure 3. The Intensional View Editor opened on the *Accept Methods* view

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. Fig. 4 shows our **Relation Browser** tool, in which relation (1) is encoded.

3.2 Check conformance

Whenever we have documented 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, fine-grained information about what went wrong will be provided, as we will see in Subsection 3.3. In that case, there are basically three 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 codify these structural regularities;

2. When the codified regularities were conceptually correct but the source code does not consistently satisfy these regularities, the developer may restructure the source code so that it does. After having modified the code, the regularity can be rechecked;

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 one should return to those strategies later to get to the heart of the problem and remove the documented deviations.

In Fig. 3 we used the 'Excludes' pane to explicitly exclude the `accept:` method on class `AbstractTerm`. Similar functionality is offered by the Relation Browser (Fig. 4) which offers the possibility to explicitly exclude or include relation tuples, or single entities from either the source or target of a relation.

3.3 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. Our toolsuite contains two dedicated tools that provide such fine-grained feedback.

The **View Inspector** (Fig. 5) 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 show the delta between the default intension and each of the alternative intensions. (In Fig. 5 there are only 2 columns but in general

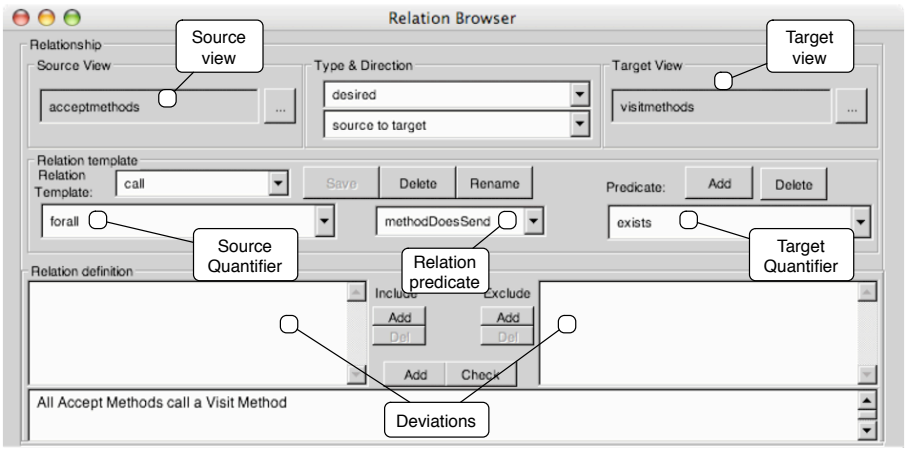

Figure 4. Relation Browser opened on the 'call' relation from *Accept Methods* to *Visit Methods*

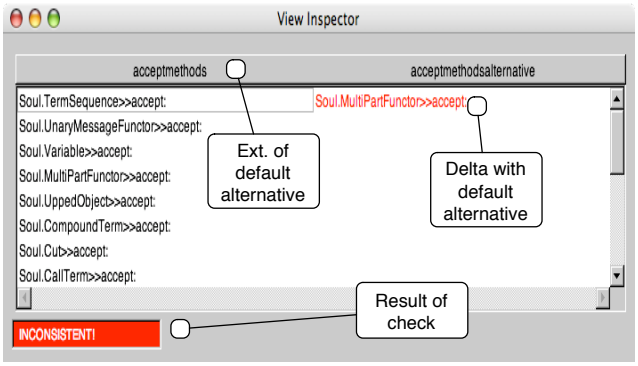

Figure 5. View Inspector on *Accept Methods*

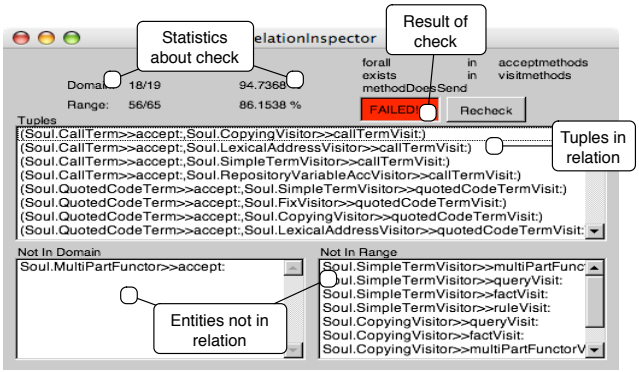

Figure 6. Relation Inspector applied to 'call' relation from *Accept Methods* to *Visit Methods*

there are as many columns as there are alternative views.) When applied to the ***Accept Methods*** view, we discover a method named `accept:` on class `MultiPartFunctor` which satisfied the default intension but not the alternative one. We discuss later how we solved this inconsistency.

The **Relation Inspector** (Fig. 6) is launched whenever an intensional relation is checked. In addition to indicating whether the relation succeeds, 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. 6), we see that the relation fails because the method `accept:`, implemented by the `MultiPartFunctor` does not call a visit method. This is the same method that causes the extensional consistency of the ***Accept Methods*** to fail. When inspecting the code of that method we see that it is in fact an abstract method. We therefore decide to explicitly exclude it from the default alternative of the ***Accept Methods*** view, just like we did with the abstract `accept:` method on `AbstractTerm`, thus resolving both inconsistencies.

3.4 Methodological aspects

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 similarity with XP testing[2] [2]:

[2]In fact, we have recently extended IntensiVE with the ability to export all intensional views and relations to be verified against the code, to a unit

- 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*.

4 Experiment: documenting and checking the structure of delfstof

We validated our approach by documenting, verifying and evolving the structural regularities of *DelfSTof* , an application written in VisualWorks Smalltalk. In another paper we reported on a case study where we applied IntensiVE to document and understand the evolution of the SmallWiki application [9].

4.1 The DelfSTof case

DelfSTof started out as a research prototype for experimenting with the technique of *formal concept analysis* to mine programs for recurring patterns [12]. Later, it was extended to accommodate for other code mining experiments.

These extensions led to a proliferation of classes (186 in the latest version) since for every new experiment we needed to define *objects* and *attributes* to be used for the concept analysis, *filters* that remove irrelevant *concepts* after the analysis, and *analyzers* that present the concepts in a format understandable to the end-user.

Because of this proliferation of classes and because symptoms of code decay started to show up, we decided to restructure the code. Before restructuring, however, we decided to document *DelfSTof* 's structural source-code regularities first.

Our main motivations for doing so were that having documented the structural regularities explicitly would help us to restructure the code, verify the impact of the restructuring on those regularities and respect the regularities in future versions of the application.

The remainder of this section illustrates by means of selective examples how we used IntensiVE to document the structural code regularities of *DelfSTof* , check them against the source code and keep new versions of the application conform to these regularities. Before doing so, the next subsection sketches the structure of the part of *DelfSTof* from which most examples in this section were selected.

4.2 The structure of DelfSTof

Fig. 7 sketches the static structure of the part of *DelfSTof* that allows it to be extended with new *objects* and *attributes* to be used by the concept analysis algorithm. Such an extension requires creating new subclasses of `ObjectCreator` and `AttributeCreator`. These classes implement the necessary methods that return a collection of objects and, respectively, a collection of attributes. To inform the concept analysis algorithm on how to use these new classes, one also needs to provide a new subclass of `ContextCreator`. This class consists of a number of factory methods which indicate the proper object creators, attribute creators, filters and analyzers to be used by the algorithm. E.g., the method `attributeCreatorClass` of this context creator class needs to be overridden so that it returns the newly created subclass of `AttributeCreator`.

DelfSTof thus imposes a number of constraints upon its possible extensions: it does not suffice to specialize the correct classes and methods, there are other structural relations between its source-code entities that need to be respected. Apart from the ones mentioned above, Fig. 7 summarizes some more of these relations that document important structural constraints for the customizers to respect.

4.3 Documenting the source-code regularities

We started out by documenting the source-code regularities of *DelfSTof* , based on our knowledge of the application, using 34 intensional views and 30 intensional relations. Instead of listing all, we limit ourselves to providing the reader with a number of interesting views and relations. For instance, we created an intensional view ***Predefined Context Creators*** that groups all predefined context creator classes of *DelfSTof* , which play a crucial role in the application. Running a specific concept analysis experiment requires creating a specific context first. Having an intensional view which contains all known classes capable of creating such a context is useful for customizers of *DelfSTof* who need to decide whether to use a predefined context creator class or rather to implement a new variant. The intensional view ***Predefined***

test suite.

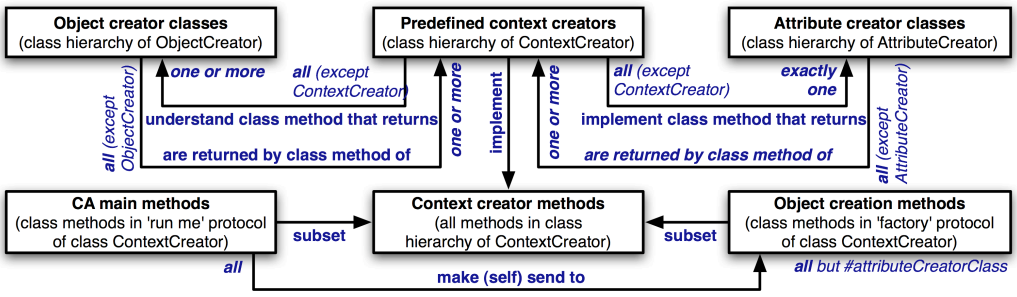

Figure 7. Static structure of part of the *DelfSTof* application

Context Creators was defined by means of the logic query: `classInHierarchyOf(?entity,[ContextCreator])` As mentioned in subsection 4.2, every concrete context creator class should have a method named `attributeCreatorClass` that returns a specific attribute creator class. Taking abstraction of the actual name of this method, we codified this structural regularity to be respected by customizers, by means of the intensional relation:

$$\forall x \in \texttt{PredefinedContextCreators} : \\ \exists! y \in \texttt{AttributeCreatorClasses} : \\ x\ classReturnsClass\ y \qquad (2)$$

where the view ***Attribute Creator Classes*** is defined analogously to the view ***Predefined Context Creators*** and where $x\ classReturnsClass\ y$ verifies if a method in class x or one of its superclasses returns class y.

Whereas the previous example codifies an obligation for application customizers, an interesting example of a prohibition is the intensional relation

$$\forall x \in \texttt{BasicAnalyzerMethods} : \\ \nexists y \in \texttt{DefaultAnalyzersAndFilters} : \\ x\ methodReferencesClass\ y \qquad (3)$$

This relation expresses that no methods of basic analyzer classes are allowed to reference any of the 'default' analyzers and filters, which represent the default behaviour of an instance of the 'Chain of Responsibility' design pattern [5]. If one of the ***BasicAnalyzerMethods*** would reference one of these classes, this would break an important structural constraint, as we will explain in more detail in section 4.5.

4.4 Checking and refining source-code regularities

Having codified a number of source-code regularities, we need to verify them against the actual source code and correct or refine any discovered inconsistencies of the source code to these regularities. Two kinds of inconsistencies can be distinguished: either our hypothesis about a structural regularity was conceptually correct but the source code did not consistently respect it, or our supposed regularity was incorrect or incomplete in which case the corresponding views or relations need to be refined.

To illustrate these two types of inconsistencies in the *DelfSTof* case we revisit intensional relation (2). When we checked its validity for the first time using the Relation Browser, it failed. By analysing the feedback we received from the Relation Inspector, we discovered two reasons why this relation failed and gained insights into how to solve the inconsistencies:

1. The view ***Predefined Context Creators***, which groups the *concrete* context creator classes, was defined as *all* classes in the hierarchy with root `ContextCreator`, including the root class itself. However, `ContextCreator` was an abstract class and did not return any of the ***Attribute Creator Classes***, as the relation required. To solve the problem we redefined the view in terms of the logic predicate `classBelow` rather than `classInHierarchyOf`, which excludes the root of the class hierarchy.

2. The view ***Predefined Context Creators*** contained a class `SoulContextCreator`. This class implements an extension to *DelfSTof* to provide an alternative way of creating contexts for the concept analysis algorithm, by using logic queries instead of factory methods. Because of this varying implementation, it did not respect the relation (2). The documented structural regularity was conceptually correct, but we discovered an inconsistency between the source code and the regularity. Instead of having been implemented as a subclass of `ContextCreator`, `SoulContextCreator` should have been implemented at the same level in the class hierarchy. Refactoring *DelfSTof* by introducing a common superclass for `SoulContextCreator` and `ContextCreator` solved this problem.

Another example of an inconsistency occurred when verifying the following intensional relation:

$$\forall x \in \texttt{AttributeCreatorClasses}: \\ \exists y \in \texttt{PredefinedContextCreators}: \\ x\ returnedBy\ y \quad (4)$$

We discovered that this relation failed because the class `AttributeCreator` was not returned by any of the predefined context creator classes. The relation encoded our hypothesis that every *specific* attribute creator class in *DelfSTof* would be used in at least one concept analysis experiment (recall that every predefined context creator class is responsible for some concept analysis experiment). We wanted to enforce this completeness constraint to avoid having too many 'retired' classes around that were no longer being used, so that the application would not get cluttered with irrelevant code. This relation however need not hold for the root class `AttributeCreator`: although it implements a default way of creating attributes and may be used as such by some experiments, there is no reason to impose that it 'must' be used in at least one experiment. By excluding the `AttributeCreator` from the source view of the relation, and thus explicitly marking it as a deviation, we were able to resolve this "violation" of the documented regularity.

4.5 Keeping source code and regularities synchronized

After having documented and verified a number of regularities in *DelfSTof*, our goal was to keep these documented regularities synchronized with the source code. In order to validate the effectiveness of our approach, we reverified the regularities with a newer version of *DelfSTof* in which, next to a number of structural changes, also a couple of customizations were added by a student who used the application.

One example of a source-code regularity that was violated in this version of delfstof was Intensional relation (2) between ***Predefined Context Creators*** and ***Attribute Creator Classes***. When checking this relation, it failed because the application had been refactored with the introduction of two intermediate abstract classes `MiningMethods` and `MiningClassesAndMethods` below the abstract class `ContextCreator`. As such, they belonged to the view ***Predefined Context Creators***. However, since they did not implement any behavior, but were introduced only as a way to regroup different kinds of context creators, they did not, and should not, satisfy relation (2). The solution we chose to bring the structural regularity up-to-date was to redefine the ***Predefined Context Creators*** view as the set of *non-abstract* classes in the hierarchy with root `ContextCreator`.

Another example is intensional relation (3) as described in subsection 4.3. As a result of new customizations which were added to the application, this relation failed as well. The analyzers and filters used in *DelfSTof* are implemented using the *Chain of Responsibility* pattern [5]. The first element of the chain is asked to analyze/filter a concept. If this element cannot handle that concept, the concept is passed to the next element of the chain. At the end of the chain a `DefaultAnalyzer` or `DefaultFilter` is placed to halt it. *DelfSTof* offers its customizers an easy interface which, when offered a sequence of analyzers/filters, automatically configures the chain with that sequence and puts the default analyzer/filter at the end of the chain. The failure of relation (3) was caused by the student who implemented the customization but was not very acquainted with *DelfSTof*. Instead of using the proper interface, he decided to construct the chain of analyzers manually. To terminate his chain he referred to the default analyzer and filter classes directly, which conflicted with relation (3) requiring that those default classes should never be referred to directly by a customizer's own analyzer methods. We fixed the problem by making the customization use the proper interface provided by the application to create the chain of responsibility.

5 Discussion and future work

The case study exemplifies the usage of the IntensiVE-toolsuite to document a program's structural regularities, to check conformance of its source code to those regularities and to offer fine-grained feedback to resolve inconsistencies when the source code does not satisfy the documented regularities. Next to having created explicit documentation of the important structural regularities in *DelfSTof*, we experienced that:

- in a number of situations, verifying conformance indicated that the structural documentation was not precise enough yet and thus needed further refinement;
- we were able to detect a number of occurrences where the application developers did not follow the coding and structuring conventions consistently or where the source code was ill-designed and prone to restructuring;
- the feedback provided by *IntensiVE* helped us understand how we could improve the documentation or the code when conflicts were detected;
- we were able to bring the documented regularities and the source code back in sync after important structural changes were made to the code and new customizations were added.

Although we were able to document the structural regularities in *DelfSTof* manually, having automated support for for *reverse engineering structural regularities* from

source code can be useful when considering large case studies of which the structure is not well-known. We are investigating several promising techniques to automate this activity, including formal concept analysis [12], clustering analysis, and inductive logic programming [21]. Complementary, a number of useful features to be integrated with our toolsuite are:

- support for *assessing the impact* of modifying the source code on the declared structural regularities.
- support for *other languages*, whether it be the language of the program of which we want to document the structure, or the query language we are using to do so. Thanks to the language independence of the underlying model, this is merely an issue of extending or reimplementing our tools for those languages. An early prototype of the *Intensional View Editor* for *Java* with support for various query languages (*TyRuBa*, *XQuery* and *XPath*) has already been developed [15], and a port of the current version of IntensiVE to Eclipse / Java is currently underway.
- A tool to visualise the documented views and relations and their conformance to the code has already been incorporated in the most recent version of IntensiVE, but was not yet used in the experiment reported on in this paper.
- support for *documenting the dynamic structure* of a program. As the model of intensional views and relations is essentially independent of the type of entities considered, what is needed to support reasoning about the dynamic structure of a program is an appropriate query language that can reason about the run-time structure of the program.

6 Related Work

In the software architecture community, a number of architecture conformance checking approaches exist that do focus on checking whether a program's source code conforms to the structure imposed by the intended architecture. Reflexion Models [14], the SAR method [7] and the NIMETA process [17] all focus on architecture reconstruction, but also support checking conformance of the source code against the intended architecture. The Software Bookshelf [4] is a collection of tools for generating software architectures from program sources and keeping this architectural documentation up to date.

Perhaps the most distinguishing feature of our approach with these approaches is that we regard the activities of documenting and checking structural source code regularities as activities that are seamlessly integrated with the development process and environment and that are carried out by the software developers themselves. The software architecture community distinguishes different kinds of architectural views such as Kruchten's 4+1 views [8]. Our approach focuses mainly on the *logical* and *development views*, which are rather implementation-oriented. It does not really support the *process view*, which captures the concurrency and synchronization aspects of the design, nor the the *physical view*, which describes the mapping of the software onto the hardware. Therefore, it would make sense to complement our approach with other approaches, like NIMETA [17], which specifically aim at modeling other architectural views too.

There exists also some research related to the model of intensional views. Most closely related are probably the models of conceptual modules [1], law-governed architecture [13] and virtual categories [19]. *Conceptual modules* are sets of lines of source code (from multiple parts of the system) that are defined intensionally by means of a regular expression and that are treated as a logic unit. Our approach however seems finer grained and expressive as intensional views are not limited to lines of code but can contain any kind of source-code entity. *Law-governed architecture* supports the declaration and enforcement of global properties of the program that may cut across the source code. As in our approach, these so-called 'law-governed regularities' are declared in, and verified with, a Prolog-like meta language. *Virtual categories* are intensionally defined sets of methods that provide support for incremental programming, by giving the developer an overview of what still remains to be done and where possible problems lie (e.g., methods that still need to be implemented and overriden methods).

In our case study, as query language to define our intensional views, as well as the binary relation predicates in terms of which our intensional relations are defined, we used the logic meta-programming language *Soul*. The model of intensional views and relations itself, however, is largely independent of the actual query language used. Using a logic meta-language has the advantage that it allows us to express powerful queries over the source code. In practice however, this expressiveness is not always needed and some queries can be rather inefficient to evaluate. Related work shows that simpler regular expression or pattern-matching languages [1, 6] may still be sufficiently powerful, yet more efficient. In fact, for most examples encountered in our case study a simpler language would have sufficed. For this reason, we are currently experimenting with other query languages like XPath, XQuery [15], regular expressions and description logics.

Finally, there exist, in the aspect-oriented software community, several tools and approaches that support reverse engineering, modelling and/or manipulation of aspects and crosscutting concerns [3, 6, 16, 18, 20]. These approaches have many similarities with ours, since intensional views too may cut across the source code, just like an aspect.

Conclusion

The main contribution of this paper is the presentation and validation of our *IntensiVE* toolsuite for documenting, checking and refining structural source-code regularities. It builds on the model of intensional views, extended in this paper with the model of intensional relations. Whereas intensional views group source-code entities that share some structural property, intensional relationships document important structural dependencies among the elements of those views. In addition to supporting conformance checking of the documented regularities to the source-code, the tools offer fine-grained feedback when the source code does not satisfy those regularities. Alongside with the toolsuite, we introduced a step-based, incremental methodology, bearing strong resemblance with XP testing. Our approach advocates the documentation and checking of structural regularities on a 'by need' basis, in order to keep the code conform to the documentation (and vice versa) as the program evolves.

Although the approach and underlying formalism are essentially language-independent, the toolsuite has been implemented in Smalltalk and reasons about Smallltalk programs using the logic meta-programming language SOUL as query language. To enable easy adoption of the toolsuite, it is relatively lightweight and has been seamlessly integrated with the VisualWorks development environment.

We validated the approach and tool suite on *DelfSTof* , an application implemented in VisualWorks Smalltalk. The results from this case study and another case study we have conducted convinced us that our framework is sufficiently expressive and mature to document, check and maintain the structural regularities of a program throughout its evolution.

References

[1] A. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the International Conference on Software Engineering ICSE'1998*, pages 64–73. IEEE Computer Society, 1998.

[2] K. Beck. *Extreme programming eXplained : embrace change*. Addison-Wesley, 2000.

[3] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. Visual separation of concerns through multidimensional program storage. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197. ACM Press, 2003.

[4] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

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

[6] W. Harrison, H. Ossher, S. M. S. Jr., and P. Tarr. Concern modeling in the concern manipulation environment. IBM Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[7] R. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[8] P. Kruchten. Architectural blueprints: The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[9] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views - a case study. *Special Issue of Elsevier Journal on Expert Systems with Applications*, 2006. Accepted for publication.

[10] 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.

[11] 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.

[12] K. Mens and T. Tourwé. Delving source-code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 31(3-4):183–197, October-December 2005 2005. To appear.

[13] N. H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.

[14] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT 1995, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[15] D. A. Ordóñez Camacho. Towards a language-independent framework for intensional views. Master's thesis, Université catholique de Louvain, 2004.

[16] H. Ossher and P. Tarr. Concern spaces: Structuring systems with hypermodules. Technical report, IBM Thomas J. Whatson Research Center, Yorktown Heights, NY, 1999.

[17] C. Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Technische Universität Wien, October 2004.

[18] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM Press, 2002.

[19] N. Schaerli and A. Black. A browser for incremental programming. *Elsevier Journal on Computer Languages, Systems & Structures*, 30(1-2), April-July 2004.

[20] S. Sutton and I. Rouvellou. Modeling of software concerns in cosmos. In *1st International Conference on Aspect-Oriented Software Development*, pages 127–133. ACM, 2002.

[21] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intensional views. *Elsevier Journal on Computer Languages, Systems & Structures*, 30(1-2), April-July 2004.