Enforcing Structural Regularities in Software using IntensiVE

Johan Brichau[a,1], Andy Kellens[b,2], Sergio Castro[a,3], Theo D'Hondt[b]

[a] *Université catholique de Louvain, Belgium*
[b] *Vrije Universiteit Brussel, Belgium*

Abstract

The design and implementation of a software system is often governed by a variety of coding conventions, design patterns, architectural guidelines, design rules, and other so-called *structural regularities*. To prevent a deterioration of the system's source code, it is important that these regularities are verified and enforced upon evolution of the system. The Intensional Views Environment (IntensiVE), presented in this article, is a tool suite for specifying relevant structural regularities in an (object-oriented) software system and verifying them against the current and later versions of the system. At the heart of the IntensiVE tool suite are (logic) program queries and the model of *intensional views and relations*, through which regularities are expressed. Upon verification of these regularities in the source code of the system, IntensiVE reports the code entities (i.e. classes, methods, variables, statements, etc.) that violate these constraints. We present IntensiVE and illustrate its application to the verification of an Abstract Factory design pattern in the implementation of a software system.

Key words: software evolution, logic meta programming, structural regularities

1. Introduction

Coding conventions, best practice patterns, idioms [1, 7], design patterns [13] and other design and stylistic guidelines have become widespread practices in the design and implementation of modern (object-oriented) software systems. Inspired by Minsky's definition of *regularities in software systems* [28], we refer to such structural guidelines as *structural regularities*. The meticulous use of regularities throughout the

Email addresses: `johan.brichau@uclouvain.be` (Johan Brichau), `andy.kellens@vub.ac.be` (Andy Kellens), `sergio.castro@uclouvain.be` (Sergio Castro), `tjdhondt@vub.ac.be` (Theo D'Hondt)

[1] Funded by the SOIB program of the "Institute for the encouragement of Scientific Research and Innovation of Brussels".

[2] Postdoctoral scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

[3] This work was partially funded by the Inter-University Attraction Poles Program of the Belgian Science Policy.

entire software life-cycle explicitly molds the software system with design and coding principles that intend to improve its quality in terms of reusability, extensibility and comprehensibility. A Visitor design pattern [13], for example, provides for extensibility of the implementation with additional operations traversing over object trees. Similarly, naming conventions render implementation concepts, such as *accessor methods*, explicit to improve the understandability of the source code, which is of specific importance in collaborative development environments. In addition, many of today's frameworks, libraries and middleware suggest a number of stylistic guidelines and impose crucial constraints on the system's design and implementation (e.g. EJB, Ruby on Rails).

In spite of their intended benefits, the consistent and meticulous application of structural regularities in the source code of a software system is often problematic. The reason for this is that most regularities are not an integrated part of the development process and programming languages of current-day implementation practices. With notable exceptions for particular kinds of regularities, such as stylistic conventions and some bad practices, which can be specified and verified using tools like CheckStyle [4] and Lint [20], the vast majority of regularities in an application remain informally defined. Without any means to document and enforce regularities in the source code, they can easily be violated, especially in subsequent evolutions of the system. In order to prevent the quality of the source code from deteriorating, it is imperative that regularities can be enforced, or at least verified, when the system evolves.

IntensiVE[4], the Intensional Views Environment [25] is a tool suite for specifying and enforcing a wide variety of structural regularities in the source code of a system. Software engineers can define regularities by means of source-code queries that gather specific source-code entities into *intensional views*, upon which constraints are imposed. Key to this technique is that it provides a means for verifying *application-specific* structural source-code regularities, much in the style of unit testing: developers can specify the regularities they deem interesting and invoke their verification at any time they desire. Typically, such structural verification is applicable following any committed evolution or maintenance activity. Upon such verification, violations of the regularities in the source code will be reported by the tool suite, allowing developers to take appropriate corrective actions.

In this article, we give a comprehensive overview on how IntensiVE is used to define and enforce structural regularities. In comparison with previous articles on the technique of intensional views [25, 26, 22], we specifically introduce the *parameterization* and *instantiation* of intensional views. This recent addition to the technique permits to parameterize the definition of a regularity such that it can be instantiated in multiple locations, both in the same and in different software projects. In the former case, it means that instances of the same regularity in the source code (such as multiple instances of the same design pattern) rely upon the same regularity definition but are verified as independent instances. In the latter case, it means that a regularity definition can be reused across different projects, eventually even facilitating the creation of reusable libraries of "regularity verification rules". In addition, while former articles

[4] `http://www.intensional.be`

have presented IntensiVE in the context of Smalltalk projects, in this article we apply IntensiVE to a Java project and demonstrate some of the new visualizations and possible customizations. Finally, we also outline IntensiVE's architecture, we present how IntensiVE itself applies to the verification of its own implementation and we discuss the technical choices that were made in its implementation.

IntensiVE is implemented in Smalltalk [14] and integrates tightly with the VisualWorks development environment [5], but can equally-well verify regularities in Cobol programs [21], and Java projects through a loose integration with the Eclipse environment. In this paper, we demonstrate the application of IntensiVE to the documentation and enforcement of a Java implementation of the Abstract Factory design pattern [13]. Section 2 elaborates on the importance of structural regularities and introduces the important constraints of the Abstract Factory design pattern. Next, Sections 3 and 4 demonstrate the definition and verification of this pattern using IntensiVE. In Section 5, we demonstrate the use of IntensiVE to express bad smells and Section 6 discusses the extensibility of the IntensiVE tool suite with a visual reporting tool for the State design pattern regularity. Section 7 gives an overview of a number of case studies that were performed using IntensiVE and, subsequently, Section 8 elaborates on the architecture and design choices taken in the implementation of IntensiVE as an extensible tool suite and as a combination of integrated Smalltalk tools. Finally, an overview of related work is given in Section 9.

2. Structural Regularities

A *structural regularity* is any decidable property of the structure of a software system that must hold true for a well-defined part of it. In addition to commonly known patterns and conventions, *application-specific* properties of the source code such as "all classes in the hierarchy of the class Command must have a name starting with prefix Command", "accessor methods must all be implemented according to the same idiom" and "entities in the presentation layer are not allowed to refer to entities in the database layer" are structural regularities.

Structural regularities play an important role in the development process. As observed by Minsky [28], the proper and meticulous use of regularities in software systems can be considered as a kind of engineering principle that aids in dealing with the inherent complexity of software systems [3]. Developers can, for example, communicate certain concepts that are only implicitly available in the source code to other developers by consistently using intention-revealing names or patterns in the source code to characterize this concept and thus make it explicit. Furthermore, regularities aid in obtaining stylistically uniform source code, leading to a more comprehensible and maintainable implementation [1]. Next to the aforementioned stylistic reasons for introducing regularities, the correct functioning of the system can depend on whether developers correctly adhere to certain regularities. For example, when making use of technology such as object-oriented frameworks, when applying design patterns, or

[5] `http://www.cincomsmalltalk.com`

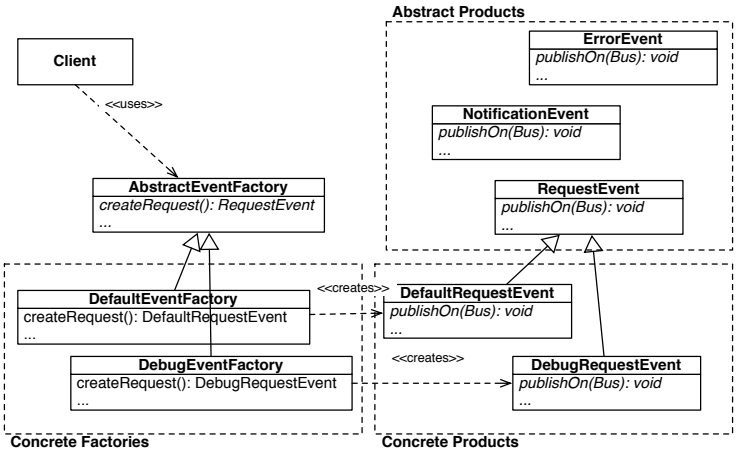

Figure 1: Abstract Factory Design Pattern

when particular platforms such as EJB are employed, developers must adhere to certain architectural or design rules imposed by these technologies. When regularities expressing such architectural or design rules are violated, this can result in erratic and incorrect behavior of a system.

2.1. Example Regularity: The Abstract Factory Design Pattern

The Abstract Factory design pattern is a widely used, yet simple example of a structural regularity in object-oriented systems. This design pattern insulates the creation of objects (products) from the client code that uses them. Its implementation consists of an abstract class that defines an interface of *product-creation* methods, and several concrete subclasses (concrete factories) that implement these methods. Figure 1 illustrates the structure of the design pattern, applied to the creation of *event* objects. Instead of creating product objects directly, clients create these objects by invoking the product-creation methods. In addition, client code must remain independent of the actual type of the product objects created by the individual concrete factories. Therefore, each *concrete product* object is a subtype of a corresponding *abstract product* type. In Figure 1, two "families" of events are distinguished: default events and debug-mode events. Each of these event-product families is a specific implementation of the set of abstract event products. An Abstract Factory pattern makes it possible to interchange the family of product objects that is created and used throughout the application by switching between different concrete factories.

Constraints

The Abstract Factory design pattern regularity imposes the following constraints over the source code:

1. The most important constraint is that product objects instantiated by the factory should not be instantiated outside the factory. If they are instantiated outside the

factory, the main reason for using the pattern is lost. It would mean that product objects of different families can co-exist at runtime, probably resulting in faulty behavior.

2. Each concrete factory needs to define product-creation methods for each abstract product. This constraint is already partially enforced because of the abstract methods in the factory superclass, imposing an implementation in each of the concrete factories. However, it is not enforced that the abstract superclass defines an (abstract) creation method for each abstract product and that the concrete factory effectively creates an instance of such a product. An evolution of the application may easily introduce a new kind of product, for which no creation method is implemented in the factory. We thus need to enforce that a product-creation method exists for each abstract product and that such a method effectively creates a new instance of that product's type.
3. In addition, the set of products created by each concrete factory must be of the same product family. In typical implementations of the pattern, this means that product objects created by one concrete factory must be of different classes than product objects created by other factories.
4. Finally, developers often use several naming conventions for the factories and the products. Factories have names that are typically suffixed with "Factory", for example. We therefore also include such naming constraints in the example.

Although each of the constraints above will most probably be adhered to when the Abstract Factory pattern is first implemented, subsequent evolutions of the software implementation can easily break one or more constraints of the regularity. In the subsequent sections, we focus how IntensiVE can be used to document these constraints and verify them with respect to the source code.

3. Expressing Regularities using IntensiVE

The heart of the IntensiVE framework harbors the model of *intensional views and relations*, through which individual constraints of a regularity are expressed. The tool suite can verify these constraints in the source code of the application and identify the source-code entities that violate them, providing feedback to the developers such that they can take appropriate corrective measures. In each of the following subsections, we explain the *definition of a regularity* using *intensional views*, *alternative views* and *intensional relations* in detail and apply them to the definition and verification of the Abstract Factory design pattern regularity.

3.1. Regularity Definitions

A *regularity definition* is the top-level concept in the model of intensional views. Each regularity definition is essentially a module that groups all the intensional views and relations that together define a single structural regularity. In the context of our example, it means that we create one regularity definition named *Abstract Factory* that will contain all the intensional views and relations needed to define the constraints of this structural regularity. This definition is illustrated in Figure 2 featuring a group of (intersecting) ellipses for each intensional view and arrows to represent the intensional

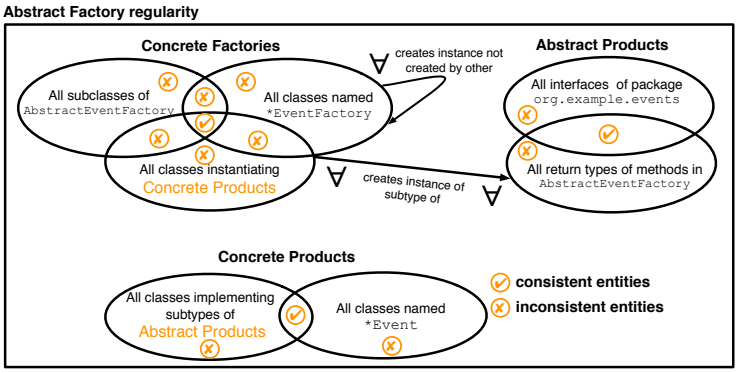

Figure 2: An illustration of the model of IntensiVE, applied to the Abstract Factory design pattern.

relations. The meaning of this visual representation and of the individual parts (views and relations) is explained in each of the corresponding subsections below.

3.2. Intensional Views

An intensional view represents a set of source-code entities (such as methods, classes, functions, and so on) in the software's implementation that make up the implementation of a concept of interest. In many cases, this concept of interest is revealed by the fact that these entities share a structural property (for example a coding convention). Therefore, typical intensional views are, for example, "all getter methods in the implementation", "all methods that invoke database operations" or "all exception handlers that only perform a logging operation". More precisely, an intensional view is a set of *tuples* of source-code entities. The idea of a tuple is that a view that represents all getter methods may also need to contain the instance variable that is referred to by the getter method. Each tuple of the view will then consist of the getter method and its corresponding instance variable. The size of the tuple and the code entities it contains, is part of the definition of the intensional view.

Intensions

The most important characteristic of intensional views is that these sets of tuples are not defined by enumeration but by means of an *intension*. Similar to set theory, an intension is an executable description that yields, upon evaluation, the set of tuples of entities belonging to the view (this set is called the *extension* of the view). Although IntensiVE is independent of the query language used, our tool tightly integrates with the logic (meta)programming language SOUL [32] (a derivative of Prolog). Its declarative source-code queries are a powerful means for the definition of intensional views and we use them throughout this article.

The use of a logic programming language to query programs has several well-established advantages [6, 32]. In imperative programming languages, programmers specify exactly *how* the solution to a problem is to be found using step-by-step algorithms. In contrast, logic programming languages allow the problem itself to be specified. The program will find a solution on its own, relying on a specific problem-solving

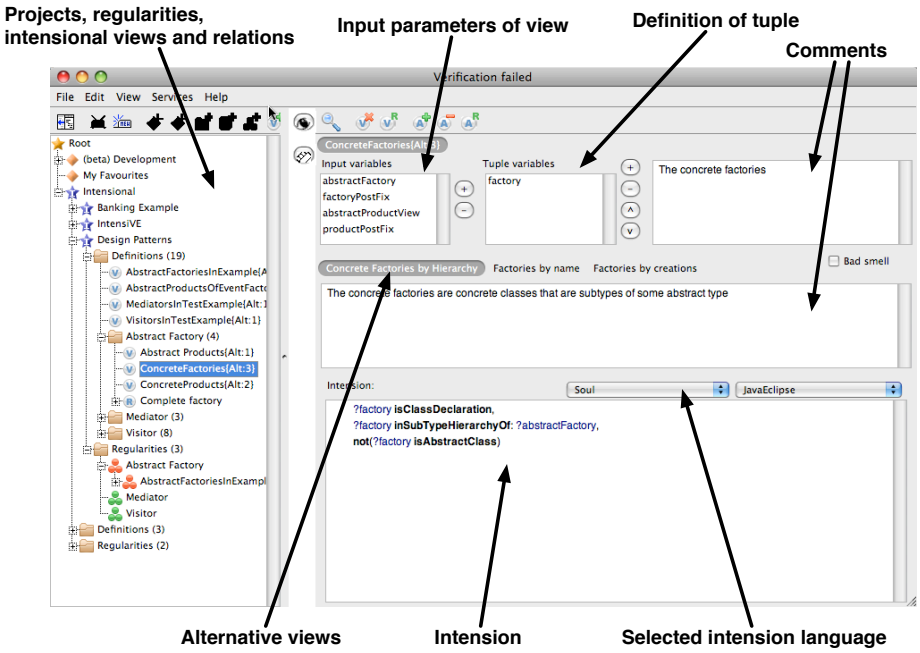

Figure 3: Definition of the *Concrete Factories* view in IntensiVE.

strategy defined by the language. In such an approach, program queries are expressed as logic conditions over the program's parts.

Consider, for example, the *Concrete Factories* intensional view that gathers all classes that implement the concept -or role- of a concrete factory. Figure 3 shows the definition of this intensional view in IntensiVE. The tuple of the view consists only of a single entity represented by the *?factory* variable and its intension is defined using the following SOUL query:

```
if ?factory isClassDeclaration,
   ?factory inSubClassHierarchyOf: AbstractEventFactory,
   not(?factory isAbstractClass)
```

This query expresses *all* conditions that a source-code entity must fulfill to be part of the intensional view. We present queries such that each condition is shown on a separate line. Also note that variables start with *?* and that the syntax of the logic predicates follows Smalltalk's messages syntax. In this simple example, the first condition expresses that an entity belonging to the view (captured by the logic variable *?factory*) must be a class declaration, which is expressed by the logic predicate `isClassDeclaration`. The following conditions specify that such a class must be a non-abstract subclass of the `AbstractEventFactory` class. The evaluation of this intension yields all classes in the source code of the system that satisfy all these conditions and, consequently, populate the *Concrete Factories* intensional view. In our example, these classes are `DefaultEventFactory` and `DebugEventFactory`.

Open Unification and LiCoR: In IntensiVE, views are populated with the actual source-code entities. This is because SOUL queries reason over a program by manipulating actual abstract syntax parse trees (ASTs). This entails that an intension can express any machine-verifiable condition over the program's structure. In addition, to hide as much as possible all the low-level peculiarities involved with direct reasoning over parse trees, SOUL's open unification technique is used to customize its reasoning process for each programming language [2]. SOUL also comes with an extensive library of logic rules, called *Library for Code Reasoning (LiCoR)*, that can be used to define an intension. Nevertheless, any developer can implement new rules because SOUL is a complete logic programming language.

In addition to the aforementioned view, the definition of the Abstract Factory regularity consists of two more views that align with the *roles* performed by the different source-code entities in the pattern's implementation. As illustrated by means of sets in Figure 2, they are the *Abstract Products* and *Concrete Products* views. In this Figure, all three views are described using multiple (intersecting) sets, each having a different *intension* shown in natural language. These are called *alternative views* and they are explained next.

3.3. Alternative Views

Often an intensional view on an implementation concept can be defined in a number of alternative and equivalent ways. In our example, the *Concrete Factories* view is not only definable as all classes in the hierarchy of `AbstractEventFactory`, but can also be defined as all classes whose name ends with "EventFactory". Furthermore, the same view can also be defined as all classes that instantiate concrete products.

In the model of intensional views and relations, such alternative intensions that define the same view are explicitly supported through *alternative views*. They are essential to express a first kind of constraint over the source-code entities contained by a view: alternative views impose that each (alternative) definition of the same view must yield the exact same set of tuples of source-code entities. This means that the source-code entities contained in the view must adhere to all conditions imposed by each alternative view. These conditions are verified by IntensiVE, which permits to detect violations of constraints that involve multiple equivalent descriptions of the same code concept. Figure 2 visually represents the idea of violations in alternative views: all classes that are situated at the intersection of the three alternative views of the *Concrete Factories* view are considered consistent entities. All other classes, that are absent from at least one alternative are considered inconsistent. For example, if a class instantiates products and is part of the `AbstractEventFactory` hierarchy, but it does not follow the naming convention, it is detected as an inconsistency.

The natural language definitions of the alternative views described above and shown in Figure 2 translate to the following two SOUL queries:

```
if ?factory classDeclarationHasName:{*EventFactory}

if ConcreteProducts(?abstractProduct,?product),
   ?factory isClassDeclaration,
   ?factory createsInstanceOf: ?product
```

The second query that is shown above also demonstrates another aspect of the tight integration of the SOUL language with IntensiVE. Its first condition retrieves the concrete product classes as they are defined by the *Concrete Products* intensional view. This is possible because the extension of an intensional view can be accessed as a predicate from within any SOUL query.

We can now summarize the essence of alternative views as follows: each alternative view should be defined as a sufficient, yet incomplete, set of conditions that a source-code entity must fulfill to implement the concept represented by the intensional view. The conjunction of all alternative intensions must yield a complete definition that all such entities must adhere to. The result is that IntensiVE will detect source-code entities that partially adhere to the complete definition of the implementation concept, and thereby detect inconsistencies in the source-code. This idea is represented in Figure 2 illustrating which parts of the intersecting sets (i.e. alternative views) contain the source code entities that are consistent with the regularity and which contain the ones inconsistent with the regularity.

Alternative views are defined in IntensiVE using a tab widget containing a tab for each alternative view, as is shown in Figure 3.

3.4. Intensional Relations

In addition to the definition of alternative views, the model of intensional views supports the definition of a second kind of constraint, namely *intensional relations*. An intensional relation can impose a quantified constraint over an intensional view (i.e. a unary relation) or between the entities contained in two intensional views (i.e. a binary relation). Any intensional relation is defined by a condition and pre-defined quantifiers. The condition expresses the actual constraints that need to be satisfied by the entities of the view(s) and the quantifiers express for which elements the condition must hold (i.e. for all, for exactly one, etc. . .). This condition can be specified using any query language but we again use SOUL to specify the condition of an intensional relation.

In our running example, the constraint that all concrete factories should be complete (i.e. that all concrete factories instantiate all types of products) is specified using a binary intensional relation. As shown in Figure 2, this relation is imposed between the *Concrete Factories* and *Abstract Products* views. The *Abstract Products* view collects the (abstract) types of all products that must be created by a factory. The relation itself is defined as follows:

$\forall$ `source` $\in$ *Concrete Factories*, $\forall$ `target` $\in$ *Abstract Products* :
factory `source` *instantiates a subtype of product* `target`

Translated to SOUL, this becomes:

$\forall$ *?source* $\in$ *Concrete Factories* : $\forall$ *?target* $\in$ *Abstract Products* :
?source.factory `createsInstanceOfType:` *?target.abstractproduct*

The logic condition verifies whether the binding of the logic variable *?factory* from the source view of the relation (i.e. *Concrete Factories*) creates an instance of the binding for the variable *?product* from the target view (*Abstract Products*). As expressed by the quantifiers, the relation is considered valid if for all possible pairs of a concrete factory and an abstract product, the above predicate holds. All other pairs are reported by IntensiVE as (possible) inconsistencies.

3.5. Regularity Instances

In a single software application, there are often multiple instances of the same structural regularity. In the context of our example, this means that besides an abstract factory for "Events", our software implementation may very well include another instantiation of the Abstract Factory design pattern, targeted at the creation of UI widgets, for example. In general, many structural regularities have multiple *instantiations* in the same software application. Since such different instantiations of the same structural regularity are subject to the same set of constraints, they should also be defined using the same set of intensional views and relations. However, since they pertain to a different part of the source code, some variations will be required. The abstract superclass of the Abstract Factory pattern, for example, differs from one instantiation of the pattern to the other.

IntensiVE explicitly supports that a single *regularity definition* describes and enforces multiple instantiations of the structural regularity in the source code. To achieve this, the regularity definition module can be parameterized with the points of variation between the different instantiations of the regularity. The idea here is that, rather than expressing an intensional view directly in terms of a particular instantiation of the regularity, all the instance-specific information is extracted from the view into input parameters. To illustrate this, let us take a look at the *Concrete Factories* intensional view. The first alternative of this view expressed that all classes in the hierarchy of the `AbstractEventFactory` are concrete factories. Rather than specifying this intension directly in terms of the specific abstract class, we define it as:

```
?factory isClassDeclaration,
?factory inSubClassHierarchyOf: ?abstractFactoryRoot,
not(?factory isAbstractClass)
```

In this intension, the actual abstract factory root class is made a parameter of the intensional view (i.e. *?abstractFactoryRoot*), and thereby also becomes a parameter of the regularity definition. The instance-specific information (i.e. the `AbstractEventFactory` class) is removed from the intension of the intensional view. This can be done for all other intensional views and relations in exactly the same way.

Figure 4 illustrates how the regularity definition of the Abstract Factory design pattern can be parameterized. In addition to the *?abstractFactoryRoot*, it is also parameterized by the *?factorySuffix*, *?productSuffix* and *?abstractProductsView* variables. The two former variables capture the actual suffix of the naming convention to be adhered by the concrete factory classes and concrete product classes, respectively. The latter variable captures the entire intension for one of the alternative views of the *Abstract Products* view. This mechanism allows to express the *Abstract Products* view in terms of a view that is defined externally to the regularity definition. Indeed, the way actual abstract products are characterized differs from one instance of the pattern to another.

For such a *parameterized* regularity definition, input arguments must be passed to describe the actual instances of the structural regularity present in the source code. Figure 4 illustrates those parameters for the instance of the pattern we have described throughout this article. Regularity definitions can be instantiated by passing actual arguments (i.e. code entities) for the parameters of the views and relations. These arguments are the extension of an intensional view, meaning that for each tuple of code

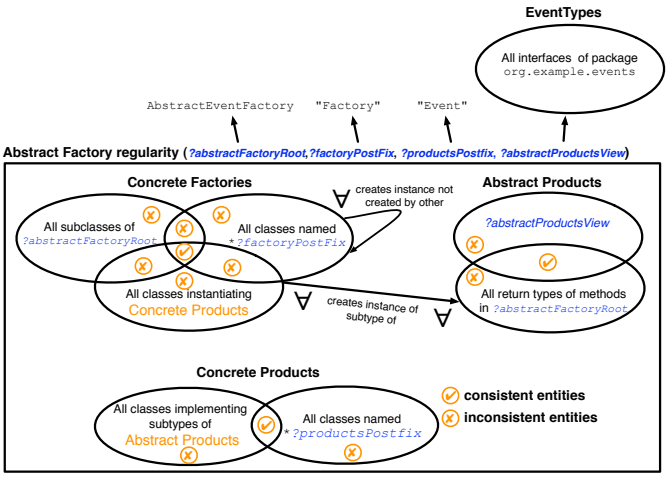

Figure 4: The parameterized regularity definition of the Abstract Factory design pattern.

entities in an intensional view, an instance of the regularity is created. In our example, such a tuple must contain the actual root class, the intended class name suffix, the root class of the products, and so on.

Figure 5 demonstrates the definition of the Abstract Factory regularity in the IntensiVE tool suite. The right-hand (definition) window features the set of views and relations that make up the regularity as well as the (input) view that defines each instance of the regularity. By selecting a mapping for the appropriate tuple variables in each view and relation, the user of IntensiVE effectively instantiates the different regularities shown in the left-hand side tree-view. A regularity instance is created for each tuple of the input view.

4. Verifying Regularities using IntensiVE

Similar to how unit testing is used to verify the functionality of a software system after evolution or maintenance, IntensiVE is used to verify the consistency of the structural regularities in the source code following an evolution. Using the intensional views and relations described in the previous section, IntensiVE can enforce the constraints of the abstract factory regularity, such as they are described in Section 2.1. For each of those constraints, we describe which intensional views and relations participate in their verification and how the consistency checking tools of IntensiVE provide developers with feedback on the violation of the constraints. Next, we show how inconsistency exceptions can be documented.

4.1. Detecting Inconsistencies

The *Concrete Factories* intensional view enforces constraints (1) and (4) of Section 2.1. Indeed, the conditions of the different alternative views express exactly these constraints. We already discussed that, if a source-code entity is not present in the

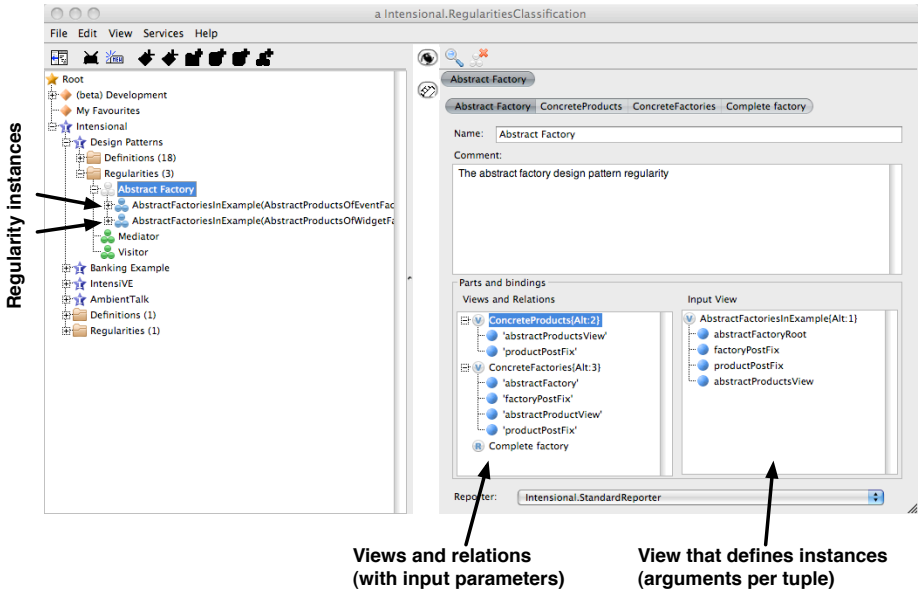

Figure 5: The regularity definition of the Abstract Factory design pattern in IntensiVE.

extension of all alternatives, it is considered to be a possible inconsistency. Figure 6 shows how IntensiVE reports on the (in)consistency of an intensional view. The rows in the table list the source-code entities belonging to the intensional view (i.e. the concrete factories). For each alternative view, a separate column is shown. The presence, or absence, of each of the source-code entities in each of the alternatives is indicated by a green (presence) or red (absence) circle in its corresponding column. For example, in the figure we can see that the *DefaultEventFactory* and *DebugEventFactory* classes are a member of all three alternative intensions. These classes thus respect constraints (1) and (4) expressed by the view. However, the Figure also shows other classes that are only present in the third alternative. These classes thus instantiate concrete products but do not follow the naming convention nor are they part of the `AbstractEventFactory` hierarchy. These classes are clearly a violation of constraint (1).

Constraint (2) is enforced by the binary relation between the *Concrete Factories* and the *Abstract Products* views, defined in Section 3.4, namely: for each abstract product, each concrete factory must implement a method that effectively creates an instance of (a subtype of) the abstract product. In exactly the same way, constraint (3) is enforced using a unary relation over concrete factories.

Figure 7 demonstrates how IntensiVE provides feedback on the verification of the first intensional relation. On the left hand side of the figure, we see a rectangle (containing smaller rectangles) representing the extension of the *Concrete Factories* intensional view, which consists of two factories (that are consistent). The second factory is indicated to be a violation since it does not instantiate 6 particular kinds of products. This result is indicated with red arrows to the elements of the second intensional view (i.e.

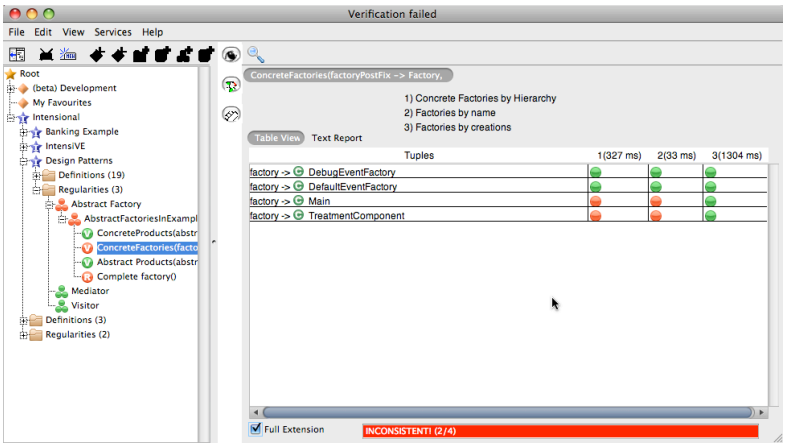

Figure 6: Verification of the *Concrete Factories* intensional view.

the *Abstract Products* view). Developers can move the mouse over these squares to find out what source-code entities these squares represent or they can divert to a textual listing of all inconsistent entities. It can also be observed in the screenshot that this reporting tool features a number of options that a developer can use to discover the violations and/or consistent elements of the target and source views as well as the respective violating tuples.

4.2. Documenting Exceptions

In many cases, detected inconsistencies are accepted by the developers as deviations to the general rule. For example, it may be acceptable that some specific class in the system creates product objects directly, i.e. without passing through a factory. Since such a class will always be detected as an inconsistency in the verification of the *Concrete Factories* view, IntensiVE allows developers to explicitly flag a code entity as an exception to the rule. In the case of alternative views, a code entity can be marked as included or excluded from a particular alternative view. Such exceptions are persistently stored and actually become part of the definition of the view itself. As a result, it will no longer be detected as an inconsistency in verifications of the view over later versions of the system but it will be reported as an exception to the general rule. This is of importance since IntensiVE is designed to support the verification of structural regularities throughout the evolution of a software application, and having the same (irrelevant) inconsistencies pop up at every verification would be undesirable.

The screenshot in Figure 8 demonstrates how such exceptions are shown when IntensiVE reports the result of the verification of the *Concrete Factories* view. Following the verification shown in Figure 6, the developer has marked the `Main` class as an exceptional inclusion in the first two alternative views (a green tick symbol in the red bullet). The result is that only the class `TreatmentComponent` is still an explicit violation.

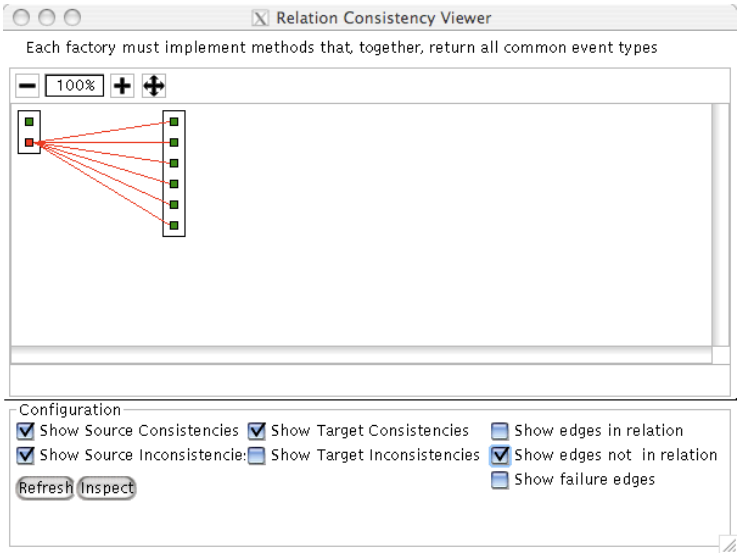

Figure 7: Verification of the binary intensional relation expressing completeness of each factory.

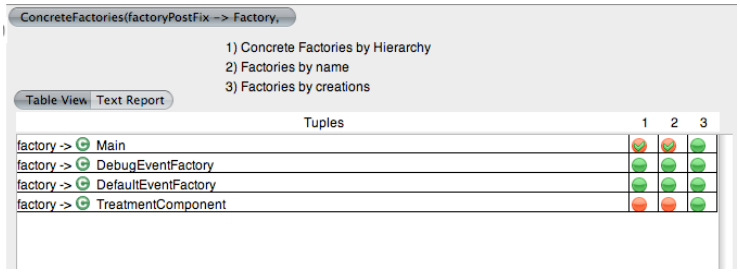

Figure 8: The Main class is documented as an exception to the general rule of the *Concrete Factories* view.

5. Support for Bad Smell and Bug Detection

Many code smells and bug patterns are much alike structural regularities, except that they are *undesirable* properties of the code. IntensiVE explicitly supports the creation of intensional views that expose such undesired structural properties. From within the user interface, an intensional view can be marked as a view for a "bad smell". This implies that *any* entity contained in the view is reported as a possible inconsistency.

For example, subtle errors can occur in Java when a constructor calls a non-final method of its class. In particular, an error occurs when the called method is overridden in a subclass and it references instance fields. These fields have not yet been initialized by the constructor of the subclass and thus contain the default initialization values, which is often an unexpected result. Although this bug is not very common, knowing that it exists in the code can save valuable time. Therefore, we define an intensional view using the following query that gathers the classes *?class*, their constructors *?constructor*, the called method(s) *?aMethod* and instance fields *?var* involved in the potential bug pattern. In summary, the query will find constructors that (transitively) invoke a method that is defined in the same class or any of its subclasses and which reads but does not write to a field defined on the same class.

```
1        ?class isClassDeclaration,
2        ?class definesConstructor: ?constructor,
3        ?constructor callsTransitiveOnSelf: ?aMethod,
4        ?subclass definesMethod: ?aMethod,
5        ?subclass isSubClassOf: ?class,
6        ?subclass definesVariable: ?var
7        ?aMethod reads: ?var,
8        not(?aMethod writesTo: ?var)
```

SOUL queries provide a versatile means to express bad smell and bug detectors. For example, note that the above query is, in comparison to bug-finding tools such as FindBugs [19], not restricted to only detecting calls to non-final methods but also verifies that the called methods actually reference an instance field. Moreover, a user of IntensiVE can further fine-tune the above query by reasoning over the method's control flow such that when the assignment to the variable occurs inside a null-checking conditional expression, the method is not detected as a bad smell [6].

6. An IntensiVE Extension: Visualized State Diagrams

Until now, we have shown how IntensiVE reports on (broken) structural regularities in terms of the source-code entities that implement or violate them. Although this works well for many regularities, often more customized feedback is desired by the development team. For example, in a particular project in which IntensiVE is applied, State design patterns [13] are used to implement state machines. These state machines are documented using state diagrams in the team's design documentation. The regularities that must be enforced in the implementation are the states and transitions as they

[6] An example of this, and more examples of bad smells, can be found on the website: http://www.intensional.be

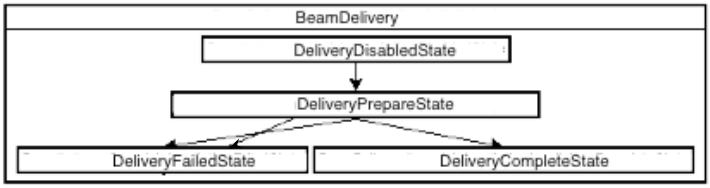

Figure 9: The State diagram of a sample State Pattern implementation.

are described in the design documentation. In other words, the developers understand these regularities in terms of state diagrams and they also desire to have IntensiVE provide them feedback at that level.

IntensiVE was specifically designed to be extensible with additional reporting tools. In this case, we implemented a reporting tool that visualizes the state diagram of each State design pattern as it is implemented in the source code. To achieve this, the classes implementing the "State" roles of the pattern and the state transitions (as implemented in the source code) were gathered into intensional views. The query that defines the view of "State" classes is relatively straightforward: it collects all subclasses of a (user-identified) abstract superclass. The view that collects all state transitions is a set of tuples of "State" classes. Each tuple thereby represents a possible state transition from one state to another state. The query that extracts this view reasons over the methods implemented on each state class (the source state) and detects the creation of instances of other state classes in the call-flow of these methods as possible destination states. This query requires only a few lines of code since (higher-order) logic predicates for iterating over the expressions in the control-flow of a method and the matching of instance-creation statements are part of the pre-defined library of logic rules.

Instead of portraying these intensional views as a collection of tuples of source-code entities, we passed on these entities to a visualization script that draws their corresponding state diagrams. Figure 9 presents such a state diagram as it is shown in IntensiVE. These diagrams reflect the actual state machine behavior as it is implemented in the source code using the State design pattern. The names of the states in the figure are the names of the classes that implement each state. At this time, developers can visually verify if the implementation corresponds to the documented state diagram. In the future, we envision that this visual reporting tool immediately reports on inconsistencies between the documented design and the implementation, which was not possible at this time because the design documentation of the state diagrams was not available in a structured format.

7. Case Studies

IntensiVE has been applied in the context of several different software projects, ranging from small academic open-source Smalltalk and Java projects to large proprietary Java and Cobol systems. In this section, we provide a brief overview of the most important case studies performed using IntensiVE, detailing context, size and results. More complete descriptions can be found on the IntensiVE website.

SmallWiki is a collaborative wiki system implemented in Smalltalk by Lukas Renggli. It is the precursor of the popular Pier wiki system. We documented some of the naming conventions, design pattern instances and design dependencies that govern SmallWiki using 27 views and relations. Particular points of interest in this case study are:

- All the various actions that can be invoked on wiki pages are implemented by means of a Command design pattern. The use of this Command pattern results in a number of interesting constraints that should be satisfied by the implementation of concrete wiki actions. First, the implementation of actions is governed by a number of naming conventions. Second, all concrete actions should follow a particular implementation strategy.
- The structure of a wiki document is represented as a tree of objects. One of the key properties of the implementation of SmallWiki is that all operations (such as visualization, storage, ...) are implemented by means of a Visitor design pattern. For this design pattern, we documented various rules that should be respected in the source code: all Visitors should be able to accept all the different kinds of wiki page nodes, all visit actions should implement the correct double dispatch protocol, and so on.

AmbientTalk is an experimental programming language to develop applications for mobile ad-hoc networks. AmbientTalk is an academic open-source project in Java for which we documented 12 regularities using 35 views and relations in IntensiVE. Here are some of the highlights:

- AmbientTalk's implementation has the concept of "native methods" which are identified by a naming convention: native method names must always start with `meta` or `base`. The naming convention of these methods is of importance because they are invoked via reflection in other parts of the implementation. We created a view containing these methods, since several constraints are imposed on them (see below).
- A design rule in AmbientTalk is that all subclasses of the class `ATObject` that implement their own "native methods" are obliged to declare to implement a Java interface that declares these "native methods". This regularity is implemented as a relation between the view of all "native methods" and a view describing these interfaces.
- The developers of AmbientTalk have established that exception handlers that handle the `SelectorNotFound` exception (part of the implementation of AmbientTalk), must invoke the `catchOnlyIfSelectorEquals` method inside the catch block to verify if the exception must be handled or propagated to another handler. This regularity is implemented as a view on all such exception handlers and a unary relation that verifies if the method invocation is found in the control-flow of the catch block.
- The signature of any native method may contain only subtypes of `ATObject`. This regularity is implemented using unary constraints on the "native methods" view.

- AmbientTalk has a notion of synchronous and asynchronous methods, each identified with a naming convention: `syncevent` and `event` respectively. The use of this naming convention is coupled to the implementation of the method itself which must use the synchronous or asynchronous message handling machinery of the AmbientTalk implementation. For this regularity, alternative views verify if the naming convention corresponds with its implementation and vice-versa.

Proprietary Java System is a relatively large Java application exhibiting a number of design regularities that have been expressed and verified using IntensiVE. This includes the Abstract Factory design pattern, very similar to the one outlined in the previous sections of this article and the state-machine extraction and visualization, also described above.

Proprietary Cobol System is a large business application for one of the larger Belgian banks. Although the development of the system started in 2005, for technical and historical reasons it was opted for to implement the system in Cobol. From the start, a lot of effort was invested in specifying a component-based design that was easily translatable into Cobol. Here is a token of the regularities we verify in this implementation:

- A Cobol system is divided in a number of different programs. In this case study, each such program implements a number of use cases that is spread out over a number of sections (cfr. procedures). In order to encode the flow of these use cases in the structure of these programs an implicit layering was introduced in the system by prefixing the names of sections. E.g. the top-level sections in a program are prefixed with an A, second-level sections with a B, and so on. In order to ensure that this layering is not violated, sections can never call another section in the program if that section's prefix precedes the prefix of the caller. For example, a section with prefix C can only call other sections prefixed C, or sections prefixed D,E, and so on.
- The documentation of the project's component-based structure is expressed using class diagrams and sequence diagrams in Rational Rose. Since it was required to assess whether the specified control flows are present in the system, we extended IntensiVE with dedicated facilities to translate these class and sequence diagrams into intensional views and relations. By providing a simple call-graph analysis over the Cobol code, we were then able to verify these sequence diagrams with respect to the actual implementation. We also provided a visual feedback of the intended sequence diagram along with calls that violate the intended sequence.

Table 7 provides some figures about the size of each case study and the time required to execute a full verification of all defined regularities on a Mac Book Pro 2.8Ghz with 4Gb Ram running Visualworks 7.6nc. However, please mind that the performance of the verification depends a lot on the kind of queries used in the defini-

Project	Code Size	Regularities	Performance
SmallWiki	108 classes, 1219 methods	6 regularities (27 views and relations)	48s
AmbientTalk	191 classes, 2347 methods	12 regularities (35 views and relations)	720s
Proprietary Java System	3468 classes, 25652 methods	8 regularities (20 views and relations)	30m
Proprietary Cobol System	1000 KLoc	9 regularities (24 views and relations)	826s

Table 1: Case Studies performed using IntensiVE.

tion of views and relations. Since each project has its custom-defined regularities, the performance of the verification of each project is highly different.

8. IntensiVE's Architecture

In this section we take a brief look at the architecture of IntensiVE and a number of regularities that pertain to it, which are enforced using IntensiVE itself.

IntensiVE's implementation does not only integrate a number of different tools, its architecture is conceived as a framework that accommodates for extension and experimentation with different source-code query languages (e.g. Smalltalk, Soul), code models (e.g. Smalltalk, Java, C, Cobol), reporting tools (e.g. separate web and desktop interfaces) and visualizations (e.g. extent of regularity in Moose polymetric views). The choice of a framework architecture and the enforcement of structural regularities are motivated by the nature of IntensiVE as the core research platform of researchers at two different universities. To explore new research directions rapidly, it was required that the tool suite could be extended by new prototype tools using minimal development effort. In addition, because many parts of the framework make assumptions about other parts of the framework (often developed by different researchers), we actively enforce these assumptions using structural regularities defined in IntensiVE itself. In combination with functional unit testing, the verification of these structural regularities provided more confidence that extensions adhere to the framework's rules and thereby prevents its implementation from deteriorating.

The Choice of Smalltalk

Smalltalk's dynamic and reflective characteristics have made it a natural choice to implement the IntensiVE tool suite. First of all, reasoning over Smalltalk programs comes naturally in such a highly reflective programing language. In addition, dynamic typing facilitated the implementation of intensional views to hold any kind of Smalltalk program fragment, Java program fragments, runtime values, or even a mixture of any of those. Also, the framework itself uses reflection to dynamically reconfigure the IntensiVE tool, depending on which plugins are loaded or not. In essence, IntensiVE reasons about its own implementation to retrieve the plugins, installed at particular

hot spots in its framework. Besides this traditional use of reflection, IntensiVE also exploits the Smalltalk reflection mechanism in the opposite way to achieve persistence of data in the form of programs. More precisely, the definition of an intensional view is stored as a Smalltalk program such that the VisualWorks storage mechanisms can make these views persistent. The open nature of the Smalltalk environment also allowed us not only to achieve a seamless integration of the IntensiVE tool suite with the IDE but also with the Smalltalk language itself. For example, we were able to extend the Smalltalk namespace mechanism to provide programmatic access to views defined in the IntensiVE tool.

An Integrated Tool Suite

In addition to the tight integration of its proper tools with the VisualWorks Smalltalk IDE, IntensiVE also integrates with the following Smalltalk tools:

StarBrowser: IntensiVE builds upon the StarBrowser [33] classification framework. This framework offers basic functionality for the creation and manipulation of classifications (i.e. sets) of objects. Since intensional views are one special kind of classification, the entire IntensiVE tool suite is conceived as an extension to the StarBrowser.

SOUL: Although it has become a standard part of IntensiVE itself, the SOUL program-query language [7] [32] is also an independent logic-based programming language, implemented in Smalltalk, that is used for meta-programming and inter-language reflection.

JavaConnect and Penumbra: JavaConnect [8] enables the seamless communication between Smalltalk and Java programs. In particular, it allows VisualWorks Smalltalk to use any Java library transparently, as if it would be a Smalltalk library. Penumbra is a Smalltalk application that uses JavaConnect to communicate with the Eclipse Java environment, which ultimately permits IntensiVE to work with the source code of any Eclipse-based (Java) project. In other words, as a meta-model for Java source code, our tool uses the Eclipse DOM/AST directly.

Moose and Mondrian: IntensiVE also integrates with the Moose [9] [10] reverse engineering environment in general and the Mondrian [27] visualization framework in particular. First of all, IntensiVE uses Mondrian to visualize the consistency of intensional relations (as shown in Figure 7). Secondly, intensional views and relations can be related to Moose models, and the associated source-code visualizations that are incorporated in Moose. This allows, for example, to visualize the extension of an intensional view into polymetric views provided by Moose [23].

Regularities in IntensiVE

The extensibility of IntensiVE relies upon the integrity of a number of structural regularities in its implementation. Evidently, we have applied IntensiVE to verify these regularities in its own implementation, which is continuously evolving. While

[7] `http://prog.vub.ac.be/SOUL`
[8] `http://www.info.ucl.ac.be/~jbrichau/javaconnect.html`
[9] `http://moose.unibe.ch`

an overview of all of these regularities lies out of the scope of this paper, we give an overview of the major categories of regularities in the current version of the tool suite[10]:

- **Design patterns** are essential to the framework's extensibility. The Command, Abstract Factory, Strategy and Visitor design patterns are among the most important regularities that are to be respected in the implementation.
- **Framework specialization constraints** are regularities imposed on (specializing) subclasses of certain classes of the framework. IntensiVE's saving mechanism, for example, relies on several naming conventions in the implementation. Another example can be found in the caching of the extension of intensional views, which can only work well if all tools that trigger the verification of an intensional view or relation also trigger the invalidation of the caching.
- **Common design constraints:** Finally, common mistakes that should avoided and rules-of-thumb that should be respected in the implementation of IntensiVE are also expressed using intensional views and relations. Typical examples of these are the fact that classes implementing an equality operation should also provide a hash function and vice versa, that overridden initialization methods must make a super send, or that for all classes in our implementation preferably there should exist a corresponding unit test class.

To document these regularities, we created 34 intensional views and 19 intensional relations that verify the regularities in IntensiVE. In order to frequently identify violations of the structural regularities in IntensiVE, the verification of the regularities occurs simultaneously with the verification of the unit tests.

9. Related Work

A substantial body of research has been devoted to supporting regularities. In general, our tool discerns itself from other approaches in that we offer an *open framework* for documenting and verifying structural regularities that is largely independent of the underlying code model that is used. Furthermore, this framework offers a declarative query language along with a vast library of logic predicates for reasoning about object-oriented programs. In what follows, we discuss five different groups of related approaches that we can discern in literature:

Code checkers: Lint [20], P^3 [8], CheckStyle [4], FindBugs [19] and many others provide developers a means to verify a wide range of applicable regularities to avoid common mistakes, bad smells, bad programming style, violations of platform-specific constraints and so on. These tools provide a *dedicated* and often highly *optimized* means to identify locations in the source code that infringe on such regularities and can provide additional support, such as (semi-)automated correction of the detected infringements. While IntensiVE does not provide the same kind of dedicated support as code checkers, our tool suite is sufficiently versatile to express the same kinds of

[10]These regularities are also present in the tool when you install it

regularities as those verified by code checkers, as exemplified in Section 5. In addition, IntensiVE is not limited to verifying the regularities supported by code checkers, but also is able to document and verify a broad scope of e.g. non-stylistic and domain-specific regularities.

Architectural and design conformance checkers: are dedicated tools that aim at verifying a high-level description of a software system (e.g. design patterns, architectural descriptions, dependencies between components, ...) with respect to the actual implementation of that system. Examples of these tools are Reflexion Models [30], Ptidej [15] and RevJava [12]. As illustrated by the Factory design pattern documentation in Section 2.1, IntensiVE can also be used to document regularities at the architectural and design level. Similar to the comparison to code checkers, IntensiVE is not specifically dedicated nor limited to these kinds of regularities but provides a general framework for documenting and verifying regularities.

Meta-programming systems: CCEL [9], Law-governed systems [28], IRC [11], GOOSE [5], CodeQuest [16], SCL [18], and many more offer developers languages for writing meta-programs that reason about programs. One application domain of these meta-program systems is the implementation of meta-programs that verify source-code regularities or that allow for imposing constraints on the source code of a system. IntensiVE is related to this group of tools in that the intension is specified by means of a meta-program, expressed using the meta-language SOUL. In contrast to these approaches however, IntensiVE aims at offering a framework at the conceptual and tool level that builds on top of the usage of a query language. Complementary to this query language, our tool offers a general metaphor for expressing structural regularities, and provides dedicated tool support for reporting violations of these regularities.

Metric-based systems: A large body of work exists that uses metrics to identify design defects or to aid in the reengineering process. Examples of such approaches are iPlasma [24], Decor [29], and RevJava [12]. While these tools aim at identifying design defects, our tool focusses on the specification and verification of *application-specific* structural source-code regularities. The goal of IntensiVE is to document the design of an application and verify it with respect to the source code; conversely, metric-based systems aim at identifying *common* defects in the design of a system by measuring its source code. The MOOSE reengineering environment [10] shares a similar goal with these approaches, namely the visualization of source-code metrics, but also offers the FAMIX meta-model for object-oriented languages that can be queried using regular Smalltalk programs. As such, MOOSE can be integrated with IntensiVE by using the FAMIX meta-model as an underlying representation of the entities that belong to intensional views. While both IntensiVE as MOOSE offer a query mechanism, our tool differentiates itself by a set of dedicated concepts (intensional views, relations, ...) — built on top of the query mechanism — for expressing and verifying structural regularities.

Pluggable type systems: Pluggable type systems [17, 31] are a means to verify additional typing constraints on the source code of a system using annotations. For example, a developer can annotate a statement indicating that the result of the statement should not be the null. The pluggable type system will — based on these annotations — verify the validity of these constraints in the source code. Most pluggable type systems come with a set of pre-defined annotations that can be used to detect common

typing mistakes in a program, along with a framework for developers to implement their own type checkers. The main difference with our approach is that IntensiVE does not rely on developers to manually annotate the source code in order to indicate where a particular constraint should apply, but rather uses declarative program queries to group sets of related source-code entities to which a constraint applies. It would be interesting to investigate how our tool framework can be used to implement the type checkers of a pluggable type system.

Conclusion

Structural regularities are omnipresent in the source-code of software applications in the form of coding conventions, design patterns, idioms, design constraints, and so on. In this paper we have presented IntensiVE, an extensible tool suite that allows developers to describe the regularities that pertain to their software application and verify these upon evolution. Although IntensiVE is tightly integrated with VisualWorks Smalltalk, it is not limited to Smalltalk projects. IntensiVE also supports the Cobol language and, through a looser integration with the Eclipse environment, the verification of Java projects. In order to define intensional views and relations, IntensiVE features a tight integration with the program query language SOUL, which offers a declarative and expressive means to reason about programs. IntensiVE can be downloaded from `http://www.intensional.be`.

References

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

[2] Brichau, J., De Roover, C., Mens, K., 2007. Open unification for program query languages. In: Astudillo, H., Tanter, E. (Eds.), Proceedings of the 16th International Conference of the Chilean Computer Science Society. IEEE Computer Society, pp. 92–101, (acceptance rate 34%).

[3] Brooks, F., April 1987. No silver bullet - essence and accidents of software engineering. Computer 20 (4), 10–19.

[4] Che06, December 2006. Checkstyle. Http://checkstyle.sourceforge.net.

[5] Ciupke, O., 1999. Automatic detection of design problems in object-oriented reengineering. In: Technology of Object-Oriented Languages and Systems (TOOLS). pp. 18–32.

[6] Cohen, J., Hickey, T. J., 1987. Parsing and compiling using prolog. Transactions on Programming Languages and Systems 9 (2), 125–163.

[7] Coplien, J., 1992. Advance C++ Programming Styles and Idioms. Addison-Wesley.

[8] Depradine, C., Chaudhuri, P., 2003. P^3: a code and design conventions preprocessor for java. Software - Practice and Experience 33 (1), 61–76.

[9] Duby, C., Meyers, S., Reiss, S., 10-13 1992. CCEL: A metalanguage for C++. In: USENIX C++ Technical Conference Proceedings. USENIX Assoc., pp. 99–115.

[10] Ducasse, S., Girba, T., Lanza, M., Demeyer, S., 2005. Moose: a Collaborative and Extensible Reengineering Environment. Franco Angeli, Milano, pp. 55–71.

[11] Eichberg, M., Mezini, M., Ostermann, K., Schäfer, T., 2004. Xirc: A kernel for cross-artifact information engineering in software development environments. In: Working Conference on Reverse Engineering (WCRE). pp. 182–191.

[12] Florijn, G., 2002. Revjava - design critiques and architectural conformance checking for java software. Tech. rep., Software Engineering Research Centre (SERC).

[13] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley.

[14] Goldberg, A., Robson, D., 1989. Smalltalk-80, The Language. Addison-Wesley.

[15] Guéhéneuc, Y., 2002. Three musketeers to the rescue – meta-modeling, logic programming, and explanation-based constraint programming for pattern description and detection. In: Workshop on Declarative Meta-Programming at ASE 2002.

[16] Hajiyev, E., Verbaere, M., de Moor, O., 2006. Codequest: Scalable source code queries with datalog. In: European Conference on Object-Oriented Programming (ECOOP). Vol. 4067 of LNCS. Springer, pp. 2–27.

[17] Haldimann, N., Denker, M., Nierstrasz, O., 2009. Practical, pluggable types for a dynamic language. Computer Languages, Systems and Structures 35 (1), 48–64.

[18] Hou, D., Hoover, J., 2006. Using SCL to specify and check design intent in source code. IEEE Transactions on Software Engineering 32 (6), 404–423.

[19] Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. ACM SIGPLAN Notices 39 (12), 92–106.

[20] Johnson, S., 1979. Lint, a C program checker. In: McIlroy, M., Kemighan, B. (Eds.), Unix Programmer's Manual, seventh Edition. Vol. 2A. AT&T Bell Laboratories.

[21] Kellens, A., De Schutter, K., D´Hondt, T., Jorissen, L., Van Passel, B., 2009. Cognac: a framework for documenting and verifying the design of cobol systems. In: Conference on Software Maintenance and Reengineering (CSMR). pp. 199–208.

[22] Kellens, A., Mens, K., Brichau, J., Gybels, K., 2006. Managing the evolution of aspect-oriented software with model-based pointcuts. In: European Conference on Object-Oriented Programming (ECOOP). No. 4067 in LNCS. pp. 501–525.

[23] Lanza, M., Ducasse, S., September 2003. Polymetric views—a lightweight visual approach to reverse engineering. In Transactions on Software Engineering (TSE) 9 (29), 782–795.

[24] Marinescu, C., Marinescu, R., Mihancea, F., Ratiu, D., Wettel, R., 2005. iplasma:an integrated platform for quality assessment of object-oriented design. In: Industrial track at the International Conference on Software Maintenance (ICSM). pp. 77 – 80.

[25] Mens, K., Kellens, A., 2006. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In: Conference on Software Maintenance and Reengineering (CSMR). pp. 239–248.

[26] Mens, K., Kellens, A., Pluquet, F., Wuyts, R., 2006. Co-evolving code and design with intensional views: A case study. Elsevier Journal on Computer Languages, Systems & Structures 32 (2-3), 140–156.

[27] Meyer, M., Girba, T., Lungu, M., 2006. Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualizaton (SoftVis). pp. 135–144.

[28] Minsky, N., 1991. Law-governed systems. Software Engineering Journal 6 (5), 285–302.

[29] Moha, N., Guéhéneuc, Y., Leduc, P., 2006. Automatic generation of detection algorithms for design defects. In: Conference on Automated Software Engineering (ASE). IEEE Computer Society Press, pp. 297–300.

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

[31] Papi, M., Ali, M., Luis Correa, T., Perkins, J., Ernst, M., 2008. Practical pluggable types for java. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 201–212.

[32] Wuyts, R., January 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Vrije Universiteit Brussel.

[33] Wuyts, R., Ducasse, S., 2004. Unanticipated integration of development tools using the classification model. Elsevier Journal on Computer Languages, Systems & Structures 30, 63–77.