Enforcing Structural Regularities in Source Code using IntensiVE

Johan Brichau*, Andy Kellens† and Kim Mens*
*Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: {johan.brichau—kim.mens}@uclouvain.be
†Vrije Universiteit Brussel
Brussels, Belgium
Email: andy.kellens@vub.ac.be

***Abstract*—The design and implementation of a software system is often governed by many different coding conventions, design patterns, architectural 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 in subsequent evolutions of the system. The Intensional Views Environment (IntensiVE), which is the subject of this demonstration, is a tool suite for documenting such structural regularities in (object-oriented) software systems and verifying their consistency in later versions of those systems.**

I. Structural Regularities

Structural regularities [1] are an apparent aspect of todays software implementations. Coding conventions, design patterns, programming idioms and architectural design constraints are only some examples of design regularities that govern the development of software systems. Maintaining these regularities in the source code of an evolving software system requires adequate documentation that is continuously monitored and verified for consistency with the source code. In the absence of such monitoring, the constant flux of evolution and maintenance tasks eventually deteriorates the system's implementation up to the point where its ability to evolve is crippled: any change to its implementation becomes too costly and time-consuming. To make matters worse, the original software design documents are hardly ever updated and become rapidly inconsistent with the systems implementation.

An Abstract Factory design pattern, for example, is a well-known and simple pattern that does impose a number of important regularities over the implementation of the entire software application. In this short description of intensional views, we limit ourselves to two important regularities: objects created by the factory should not be created outside of the factory (1) and a factory needs to define product-creation methods for each kind of product (2).

II. Intensional Views and Constraints

Intensional views, and the associated IntensiVE tool suite [2], [3], permits to construct documentation that actively verifies structural regularities by grouping source-code entities into views using program queries, and by imposing constraints over these views. By checking the validity of the views and constraints with respect to the source-code, the tool suite provides fine-grained feedback concerning inconsistencies between the documentation and the source code. IntensiVE is tightly integrated with the Visualworks Smalltalk Environment and has a looser integration with the Eclipse IDE.

An intensional view is a set of source-code entities in the software's implementation that share some structural property. Typical intensional views are, for example, "all methods that invoke database operations" or "all exception handlers that only perform a logging operation". An important characteristic of intensional views is that these sets are not defined by enumeration but by means of a *program query*. Such a program query is an executable description that yields, upon evaluation, the set of source-code entities belonging to the view. For this purpose, the IntensiVE tool suite is tightly integrated with the program query language SOUL [4] (a derivative of Prolog). A query in this language that gathers all (Java) classes `?factory` that create an instance of a class type `?product` that is a subtype of `be.intensional.Product`, is shown below:

```
?factory isTypeDeclaration,
?factory createsInstanceOfType: ?product,
?product isSubtypeOf: {be.intensional.Product}
```

To verify and enforce an actual structural regularity over the collected program entities, the tool suite allows to impose constraints over the intensional views using two different techniques:

- **An alternative view** uses an alternative program query for an already existing intensional view. Upon evaluation, this alternative view must contain the exact same set of source-code entities as the original view. IntensiVE automatically verifies the consistency of the source code by checking if the set of entities contained in the original and alternative views are equal.
- **An intensional relation** imposes conditions to be verified between the entities of one or two different views. The constraint is implemented as a combination of a query and pre-defined quantifiers. The query expresses the conditions that entities of the different views need to satisfy and the quantifiers express for which elements those conditions must hold (i.e. for all, for exactly one,...).

Figure 1 shows how inconsistencies in the source code,

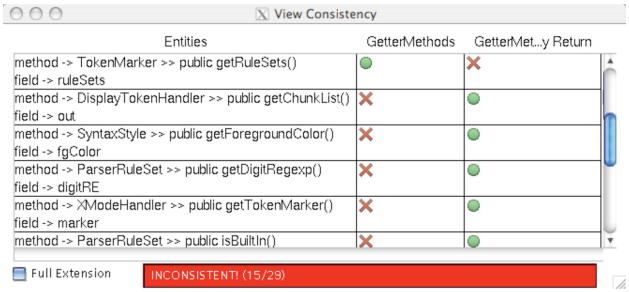

Fig. 1. Inconsistent alternative views.

detected by alternative views, are reported by IntensiVE. The inconsistency window displays a list of method entities that are missing from one of the two alternative views (displayed in columns). This screenshot concerns an intensional view named "getter methods" where one alternative view gathers methods prefixed with `get` and another alternative view gathers methods that return an instance variable. The methods are part of the implementation of the jEdit application and merely serve an illustrational purpose.

Finally, program entities that are exempted from particular regularities can be explicitly classified as exceptions to the rule by users of the tool suite.

III. Abstract Factory Example

The regularities of an abstract factory design pattern, as outlined in the introduction, can be enforced in IntensiVE using the following views and relations. Upon the verification of these views and relations, IntensiVE will report all source-code entities that violate the imposed constraints.

1) The "Concrete Factories" view gathers all concrete factories implemented in the system. The query that defines this view collects all subclasses of the abstract factory superclass. Regularity (1) is enforced by defining an alternative view that gathers all classes in the system whose methods contain instance creation statements for classes identified as products. Such product classes are identified using another intensional view:
2) The "Products" view groups all classes that must be created by factories. It gathers all classes for which an instance creation statement is found inside the implementation of the concrete factory classes. An alternative view is defined based upon application-specific rules such as a coding convention or the implementation of a (Java) interface by all product classes. Thereby, regularity (2) is partially verified.
3) An intensional relation ensures that for each product in the "Products" view, at least one concrete factory of the "Concrete Factories" view must implement a method that effectively creates an instance of (a subtype of) the product. This relation completes the verification of regularity (2) and will report all products that are not created by a concrete factory.

A more precise verification of the abstract factory design pattern, featuring addition regularities, will be outlined during the tool demonstration.

IV. Related Work

IntensiVE is related to a large number of code verification tools such as Lint, P^3, CheckStyle, FindBugs, Reflexion Models[5], Ptidej[6], CCEL[7], IRC[8], SCL[9] and many more. All these tools provide a *dedicated* and sometimes highly *optimized* means to identify infringements on regularities such as commonly-made mistakes, bad smells, bad programming style, violations of platform-specific constraints and so on. In contrast, IntensiVE does not provide the same kind of dedicated support as individual code checkers but it is sufficiently versatile to express the same kinds of regularities. In addition, and more importantly, IntensiVE is specifically designed to document and verify non-stylistic, application-specific regularities.

Acknowledgments

Johan Brichau is funded by a "FIRST" post-doctoral research grant provided by the Région Wallonne, Belgium. Andy Kellens is funded by a post-doctoral research grant provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

References

[1] N. Minsky, "Law-governed systems," *Software Engineering Journal*, vol. 6, no. 5, pp. 285–302, 1991.
[2] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Elsevier Journal on Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.
[3] K. Mens and A. Kellens, "IntensiVE, a toolsuite for documenting and checking structural source-code regularities," in *Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 239–248.
[4] R. Wuyts, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Vrije Universiteit Brussel, January 2001. [Online]. Available: http://prog.vub.ac.be/Publications/2001/vub-prog-phd-01-01.pdf
[5] G. Murphy, D. Notkin, and K. Sullivan, "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, 1995, pp. 18–28.
[6] Y. Guéhéneuc, "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*, 2002.
[7] C. Duby, S. Meyers, and S. Reiss, "CCEL: A metalanguage for C++," in *USENIX C++ Technical Conference Proceedings*. USENIX Assoc., 10-13 1992, pp. 99–115.
[8] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer, "Xirc: A kernel for cross-artifact information engineering in software development environments," in *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2004, pp. 182–191.
[9] D. Hou and J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.