Science of Computer Programming 75 (2010) 232–246

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Enforcing structural regularities in software using IntensiVE

Johan Brichau [a,*], Andy Kellens [b], Sergio Castro [a], Theo D'Hondt [b]

[a] *Université catholique de Louvain, Belgium*

[b] *Vrije Universiteit Brussel, Belgium*

ARTICLE INFO

Article history:
Received 1 December 2008
Received in revised form 13 November 2009
Accepted 16 November 2009
Available online 20 November 2009

Keywords:
Software evolution
Logic meta-programming
Structural regularities

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.

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 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.

* Corresponding author.
E-mail addresses: johan.brichau@uclouvain.be (J. Brichau), andy.kellens@vub.ac.be (A. Kellens), sergio.castro@uclouvain.be (S. Castro), tjdhondt@vub.ac.be (T. D'Hondt).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.11.005

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,[1] 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 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,[2] 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 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

1 http://www.intensional.be.

2 http://www.cincomsmalltalk.com.