Chapter VI

SPINE:
Language for Pattern Verification

Alex Blewitt, Edinburgh University, UK

Abstract

Patterns are often described in terms of concrete examples in specific programming languages in catalogues (Gamma, Helm, Johnson, & Vlissides, 1995). The description is worded such that a practitioner in an object-oriented programming language will be able to understand the key points of the pattern and translate it into a programming language of their choice. This abstract description of patterns is well suited for intelligent readers, but less suited for automated tasks that must process pattern information. Furthermore, the way in which the pattern information is encoded is often strongly influenced by the type of processing that is being performed on the pattern. In this chapter, the SPINE language will be presented as a way of representing Design patterns in a suitable manner for performing verification of a pattern’s implementation in a particular source language. It is used by a proof engine called HEDGEHOG, which is used to verify whether a pattern is correctly implemented.
Background

Patterns are often designed into a system at an early stage, often before any code has been written. The use of patterns in systems software is well known and the architect will have an appreciation of the benefits of a particular pattern when designing a system in the first place. However, errors in realising these patterns can creep into the system by two means; either an error in transcribing the architect’s design into code, or a subsequence maintenance change that violates one of the pattern’s assumptions.

Given that a pattern is an abstract concept, language compilers and other verification tools do not necessarily know that a particular collection of code exhibits a pattern. Automated testing harnesses (such as JUnit and TestNG) tend to be utilised to check the behaviour of code, rather than its implementation. A pattern normally does not have a specific behaviour that can be checked directly, and indeed, two pieces of code may have identical external behaviour but be implemented in completely different ways (using different patterns, or no pattern at all). As a result, there is an opportunity for a different type of tool that can verify whether a pattern exists by analysis of the code instead of the code’s behaviour.

The goal of Spine is to provide a language that can represent a Design pattern’s constraints such that they can be proven with an automated proof tool Hedgehog (Blewitt, 2006). This language may also be useful for reasoning about patterns in other ways; for example, the same declarative specification of a pattern could be used to instantiate a pattern, or guide a refactoring tool toward the introduction of a pattern into existing code, although these are not specifically addressed by this chapter. The section on further work discusses these ideas in more detail.

Why a New Language?

Other languages already exist that can be used to describe patterns. Catalogues often present summaries in UML (Fowler & Scott, 2003), and of course the target language (Java, C++, or Smalltalk) is often used to give an example of a pattern being implemented. Why not just use one of the existing languages to describe patterns?

The problem is that the target language is sufficiently flexible to allow a pattern to be realised in a number of ways. A Singleton’s instance does not need to be stored in a variable called default, nor does Observer’s event mechanism need to be called notify(). Thus, a specific implementation of a given Design pattern cannot be compared on a like-for-like basis between a known example in order to provide differences. Similarly, regardless of the target language used, UML cannot be used to describe an infinite set of pattern instances because the language is not designed for that purpose. If there were some higher-level metaUML, then perhaps this would be appropriate, but the UML itself does not provide this.

Other languages have been introduced in the past. Amnon Eden’s LePUS (1998, 2000) provided such a metamodelling language in order to describe relationships between classes, and then relationships between those relationships as well. The modelling language was graphical—like UML—because more information can be encoded in a graphical diagram than text. As with any graphical modelling language, the notation used to represent relationships between classes, both from an object inheritance point of view, but also in terms of the
12 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the product's webpage:

www.igi-global.com/chapter/spine-language-pattern-verification/8154?camid=4v1


Recommend this product to your librarian:

www.igi-global.com/e-resources/library-recommendation/?id=1

Related Content

Design of Reconfigurable Architectures for Steganography System

www.igi-global.com/chapter/design-of-reconfigurable-architectures-for-steganography-system/168905?camid=4v1a

Counting People Using Blobs and Contours
Shafraz Subdurally, Devin Dya and Sameerchand Pudaruth (2013). International Journal of Computer Vision and Image Processing (pp. 1-16).

www.igi-global.com/article/counting-people-using-blobs-and-contours/87247?camid=4v1a

Metal Artifact Reduction: A Problem of Tremendous Importance in Medical Imaging
Shrinivas D. Desai and Linganagouda Kulkarni (2016). Handbook of Research on Emerging Perspectives in Intelligent Pattern Recognition, Analysis, and Image Processing (pp. 296-316).

www.igi-global.com/chapter/metal-artifact-reduction/141640?camid=4v1a
Word-Level Multi-Script Indic Document Image Dataset and Baseline Results on Script Identification