Science of Computer Programming 80 (2014) 211–222

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Automated verification of design patterns: A case study

Jon Nicholson [a], Amnon H. Eden [b,*], Epameinondas Gasparis [b], Rick Kazman [c,d]

[a] School of Computing, Engineering and Mathematics, University of Brighton, UK
[b] School of Computer Science and Electronic Engineering, University of Essex, UK
[c] Software Engineering Institute, Carnegie-Mellon University, USA
[d] University of Hawaii, USA

HIGHLIGHTS

- The problem of conformance of evolving programs to design decisions motivates this paper.
- A leading example demonstrates using a tool to fully automate conformance checking.
- "Java's AWT package conforms to the Composite pattern" is formalized and proven.
- A tool for specifying visually the pattern and verifying conformance of Java code is shown.

ARTICLE INFO

Article history:
Received 27 September 2012
Received in revised form 21 May 2013
Accepted 26 May 2013
Available online 2 June 2013

Keywords:
Object-oriented design
Modeling and specification
Automated verification
Visual languages
Design description languages

ABSTRACT

Representing design decisions for complex software systems, tracing them to code, and enforcing them throughout the lifecycle are pressing concerns for software architects and developers. To be of practical use, specification and modeling languages for software design need to combine rigor with abstraction and simplicity, and be supported by automated design verification tools that require minimal human intervention. This paper examines closely the use of the visual language of Codecharts for representing design decisions and demonstrate the process of verifying the conformance of a program to the chart. We explicate the abstract semantics of segments of the Java package java.awt as a *finite structures*, specify the Composite design pattern as a Codechart and unpack it as a set of formulas, and prove that the structure representing the program satisfies the formulas. We also describe a set of tools for modeling design patterns with Codecharts and for verifying the conformance of native (plain) Java programs to the charts.

1. Introduction

Software systems are some of the most complex artifacts ever produced by humans [1,2]. Managing complexity is one of the central challenges of software engineering. Lehman's second Law of Software Evolution [3] suggests that complexity further arises when programs are maintained in a continuous state of flux, a situation which is true for many software systems. These concerns require specification and modeling languages for software design to combine abstraction mechanisms with rigor and parsimony. In addition, practitioners find it easier to use a visual notation to articulate design decisions. Therefore, accurate specification of software design and the means for checking conformance of native source code thereto are primary concerns. Given the complexity of design verification and the frequency by which they need be carried out, practitioners also need tools that automate and report any conflicts between design and implementation at the click of a button. However, these needs have so far been difficult to reconcile in practice. Proving and preserving the conformance of

* Corresponding author. Tel.: +44 1206872677.
E-mail addresses: a@eden-study.org, eden@essex.ac.uk (A.H. Eden).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.05.007

a program to its design is largely an unsolved problem. The result is often a growing disassociation between the design and the implementation [4].

The language of Codecharts [5], LePUS3, is a formal and visual design description language tailored to meet these concerns. It supports the representation of structural information about object-oriented design motifs, programs of any size, and frameworks. The Two-Tier Programming Toolkit [6] was developed to demonstrate the feasibility of specification and automated verification of Codecharts and to test it in practice.

This paper presents a case study which highlights the process by which practitioners can use Codecharts to represent design patterns and supporting tools to verify design conformance of Java programs fully automatically. In the remainder of this section we discuss other attempt at this problem. We also present the definition of the Composite pattern, and the java.awt package in version 1.5 of the Java Standard Development Kit. In Section 2 we present an informal hypothesis (Hypothesis A) about the conformance of the java.awt package to the Composite design pattern [7]. This hypothesis is gradually rendered formal (Hypothesis E) through Sections 3 and 4. In Section 5 we present a logic proposition that formalizes our hypothesis and prove it. In Section 6 we present a tool that fully automates the verification process and reports any violations of the respective design decisions. Section 6 concludes with a brief discussion on the results of a pilot study that evaluated the tool. This study showed that the tool improved the ability of participants to detect violations of design specifications.

1.1. Related work

There exist numerous attempts at formal specification languages for design patterns. Some preliminary work has also been published on tools which verify that such specifications were properly implemented in source code. The following set of criteria guides our analysis of these languages:

1. **Object-oriented**: the language must be suitable for modeling and specifying the building-blocks of object-oriented design patterns.
2. **Generic**: the language must have the ability to represent design motifs such as patterns in terms of generic 'participants' [7] (also *placeholders* or *roles*) as distinguished from concrete implementation artifacts (e.g. classes, methods etc.). A tool should support the specification of many patterns rather than being hard-coded to verify specific patterns.
3. **Implementation independent**: specifications in this language should not be bound to a specific programming language or to any specific dialects.
4. **Visual**: specifications should be represented visually and created using a visual editor for ease of use.
5. **Parsimonious**: the language must have the ability to represent complex design statements parsimoniously, using a small vocabulary.
6. **Rigorous**: the language need be mathematically sound and axiomatized such that all assumptions are articulated explicitly and precisely.
7. **Decidable**: the language is restricted to expressing properties and relations whose satisfaction can be established statically ('structural statements'), which ensures that specifications are automatically verifiable at least theoretically.
8. **Automatically verifiable**: specifications in this language must allow fully automated design verification against programs in their native, uninstrumented form (source code).

Several formal notations for specifying design patterns are described in [8]. Most of the contributions in this volume do not describe tools for automated verification, and base their notations on UML. UML is a popular visual object-oriented design description language, a powerful and expressive collection of notations [9] suitable for many common software development tasks. For example, DPML [10], which is based on UML, is capable of representing both programs and design patterns. UML, however, does not meet all of our above criteria. In particular, Fowler [11] tells us that "no formal definition exists of how UML maps to any particular programming language". In other words, UML as a specification language does not meet the criteria of being rigorous and automatically verifiable. Blewitt [12] adds that "UML cannot be used to describe an infinite set of pattern instances because the language is not designed for that purpose". Thus UML dialects that do not introduce variables for the representation of generic participants do not meet the criterion of genericity.

Many specification languages whose semantics are defined in terms of UML and tools depending on such representations face similar issues. The DEMIMA framework [13] can check the conformance of source code to design patterns specified in the Pattern and Abstract-level Description Language (PADL), which translates UML diagrams to a constraint-based language. The authors represent such constraints using a Java data structure implemented using the Ptidej tool suite. However they do not describe a tool that can create PADL models from visual specifications. The Pattern Specification Language (PSP) [14,15] articulates design patterns in precise and generic terms. [16] define the manual process of design verification of instances of the Visitor pattern specified in PSP. However, although PSP formalizes a subset of UML, it is symbolic and not visual. LAMBDES-DP, described in [17], is a tool that detects instances of design patterns in UML models and formalized in GEBNF (Graphically Extended BNF) but not in source code.

Spine [12] is a language outside of the UML family. It is a formal object-oriented language for representing design patterns in the logic programming language PROLOG. Specifications written in Spine are automatically verifiable using its associated tool, Hedgehog [12]. However, Spine is not a visual language, and "all of the Spine predicates are tightly focused