Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 231–249
Science of Computer Programming
www.elsevier.com/locate/scico

Sequences as a basis for pattern language composition

Ronald Porter[a,*], James O. Coplien[b], Tiffany Winn[a]

[a] *School of Informatics and Engineering, Flinders University of South Australia, PO Box 2100, Adelaide, South Australia 5001, Australia*
[b] *Vrije Universiteit Brussel, c/o North Central College, 30 N. Brainard Street, Naperville, IL 60540, USA*

Received 12 November 2003; received in revised form 10 August 2004; accepted 6 September 2004
Available online 13 December 2004

Abstract

Pattern languages have begun to appear and mature as a presentation of the structures and processes that support the building of complex software systems. A pattern language describes how to compose structures in a particular domain such as telecommunications, client–server architecture, or object-oriented programming, to achieve system-level architectures that are greater than the sum of their parts. A problem lurks on the horizon: How do you compose patterns from multiple domains—from multiple pattern languages—in a single system? For example, today there is nothing other than an ad hoc approach to combining the pattern languages for telecommunications and for object-oriented design to build object-oriented telecommunications systems from the respective pattern languages. To understand the solution to this dilemma, it pays to examine sequences: an important aspect of pattern application that is often overlooked. Sequences are the loci of concern about interleaving patterns, whether from a single pattern language or multiple pattern languages. Sequences are critical because pattern languages represent long-term archives of the rhythms of critical, recurring sequences.

Keywords: Sequence; HOPP; Idioms; AOP; Aspects; Telecommunications

* Corresponding author.
E-mail address: ron.porter@infoeng.flinders.edu.au (R. Porter).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.11.014

1. Introduction

Modularity appears to be dead. The focus of contemporary design discourse is typified by such non-modular notions as aspects, patterns, and features. Aspects are a grouping of code related to some software feature whose implementation cuts across the original system partitioning. Patterns are structural relationships between parts of a system on many scales; they combine architectural relationship, structure, and process in ways that go beyond modular approaches. Features are collections of marketable functionality which, though they would be attractive as self-contained physical packages, almost always cut across any reasonable physical partitioning of the architecture.

Aspects and patterns are both examples of design aids that cut across existing, recognized modules. And yet, the irony is that these examples are themselves modular in some sense. Alexander's early work viewed patterns as encapsulations of forces [2, p. i], as design modules that allowed one to reason about a limited set of forces in relative isolation. Both patterns and aspects identify key elements of complex system composition at the conceptual level. Aspects, patterns, and features are thus themselves subject to questions of composition. For example, how might one compose multiple aspects on top of a single system? How does one compose features? This problem is as yet largely unsolved.

An ordered composition of patterns leads to a system. The structural relationships between patterns are encoded in larger design structures called pattern languages. The term "language" appeals to the notion of a generative grammar that can create any system in a given genre. Much of the software community's original interest in patterns came from their ability to deal with issues that, from an object-oriented perspective, reflected cross-cutting.

Yet, even patterns are not immune from the cross-cutting problem. Most interesting systems are not of a single, pure genre. For example, a telecommunications system may use communication design pattern languages such as those given by Hanmer [1] and Meszaros [21] for fault tolerance and system capacity, respectively. The structural ordering is well defined within each of these pattern languages. Yet what does one do if one wants to build a fault-tolerant system that deals with capacity issues? How does one mix patterns from multiple pattern languages? The problem can become arbitrarily rich: consider adding yet a third pattern language such as Coplien's language for structuring C++ inheritance hierarchies [11], presuming that the system is to be written in C++ using an object-oriented programming style.

The fundamental problem of combining patterns of different genres is not one of spatial dependency, but of the temporal ordering of pattern application. This means that combining pattern languages is more temporal than structural. While expert telecommunications designers have long known instinctively how to combine such pattern languages for a given system and context, articulating that knowledge for less expert designers is a very difficult problem. For the inexpert designer, who may be using particular pattern languages for the first time, the languages offer no help as to how to address the problem of temporal ordering of pattern application. Addressing that problem is the focus of this paper.

The structure of the paper is as follows. Section 2 defines pattern languages. Section 3 further motivates the need for composition. Section 4 talks about sequences, and Section 5 discusses pattern languages as encodings of recurring rhythms of sequences. Section 6